INTEGRATED CIRCUITS

16-bit 80C51XA Microcontrollers (eXtended Architecture)

XA
16-bit
8-bit

1995

DATA HANDBOOK IC25

Philips
Semiconductors

PHILIPS

QUALITY ASSURED

Our quality system focuses on the continuing high quality of our components and the best possible service for our customers. We have a three-sided quality strategy: we apply a system of total quality control and assurance; we operate customer-oriented dynamic improvement programmes; and we promote a partnering relationship with our customers and suppliers.

PRODUCT SAFETY

In striving for state-of-the-art perfection, we continuously improve components and processes with respect to environmental demands. Our components offer no hazard to the environment in normal use when operated or stored within the limits specified in the data sheet.

Some components unavoidably contain substances that, if exposed by accident or misuse, are potentially hazardous to health. Users of these components are informed of the danger by warning notices in the data sheets supporting the components. Where necessary the warning notices also indicate safety precautions to be taken and disposal instructions to be followed. Obviously users of these components, in general the set-making industry, assume responsibility towards the consumer with respect to safety matters and environmental demands.

All used or obsolete components should be disposed of according to the regulations applying at the disposal location. Depending on the location, electronic components are considered to be 'chemical', 'special' or sometimes 'industrial' waste. Disposal as domestic waste is usually not permitted.

16-bit 80C51XA (eXtended Architecture) Microcontrollers Data Handbook

CONTENTS

		page
SECTION 1	GENERAL INFORMATION	5
SECTION 2	XA USER GUIDE	29
SECTION 3	XA FAMILY DERIVATIVES	315
SECTION 4	FUTURE DERIVATIVES	405
SECTION 5	APPLICATION NOTES	411
SECTION 6	DEVELOPMENT SUPPORT TOOLS	477
SECTION 7	PACKAGE INFORMATION	503
APPENDIX A	DATA HANDBOOK SYSTEM	512

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS
Philips Semiconductors and Philips Electronics North America Corporation Products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation Product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation Products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from such improper use or sale.

Philips Semiconductors and Philips Electronics North America Corporation register
eligible circuits under the Semiconductor Chip Protection Act.

© Copyright Philips Electronics North America Corporation, 1995

All rights reserved.
Printed in U.S.A.

Preface

XA Microcontrollers from Philips Semiconductors

Philips Semiconductors offers a wide range of microcontrollers based on the 8048, 80C51, 68000, and now the XA architectures. The XA is a new architecture that was developed by Philips Semiconductors in response to the market need for higher performance than what can be obtained from the 8-bit 80C51 and retained compatibility with the 80C51 designed-in architecture. The XA successfully addresses both of these needs. It is compatible with the 80C51 at the source code level. All of the internal registers and operating modes of the 80C51 are fully supported within the XA, as are all of the 80C51 instructions. Yet compatibility with the 80C51 has in no way hindered the performance of the XA, a very high performance 16-bit architecture. The XA's performance is 3 to 4 times faster than that of the most popular 16 bit architectures and 10 to 100 times faster than the 80C51.

If you use or are familiar with the 80C51 and need higher performance, the XA is the architecture for you. You will find it very easy to understand. Rather than having to learn its programmer's model, you will find that you already know it, and, better, are very familiar with it. You will be able to focus on the enhanced features of the XA and quickly move your design to much higher performance. You will also notice that the features on the XA, in many cases, exceed what you need today. We have designed the XA so that it will meet your needs not only today but well into the future; you will not need to look for another architecture for many years to come.

As Philips Semiconductors has done with the 8048 and 80C51, we will develop the XA into a broad family of derivatives. Advance information has been included in this handbook that covers the first two of these. It is our plan to introduce 3 to 4 XA derivatives in 1996 and 5 to 8 per year after that. In addition to this, we will continue to move the XA into Philips Semiconductors' most advanced processes and we have plans to increase the clocking frequency of the architecture to over 100MHz (greater than 30MIPS execution rate).

Philips Semiconductors offers you one of the industry's widest selections of microcontrollers. The XA architecture is an extension of this strategy that gives you the ability to easily upgrade your designs to very high performance with the only 16-bit, 80C51-compatible microcontroller available on the market.

Section 1

General Information

CONTENTS

Contents 7
Ordering information 10
Product Status 11
XA tools linecard 12
Microcontroller bulletin boards 13
Philips Fax-On-Demand System 14
CMOS and NMOS 8-bit microcontroller family 15
CMOS 16-bit microcontroller family 23
80C51 microcontroller family features guide 24
Handling MOS devices 28

CONTENTS

IC25: 16-bit 80C51XA (eXtended Architecture) Microcontrollers

Preface 3

Section 1 – General Information

Contents 7
Ordering information 10
Product Status 11
XA tools linecard 12
Microcontroller bulletin boards 13
Philips Fax-On-Demand System 14
CMOS and NMOS 8-bit microcontroller family 15
CMOS 16-bit microcontroller family 23
80C51 microcontroller family features guide 24
Handling MOS devices 28

Section 2 – XA User Guide

- 1 **The XA Family – High Performance, Enhanced Architecture 80C51-Compatible 16-Bit CMOS Microcontrollers** 31
  - 1.1 Introduction 31
  - 1.2 Architectural Features of XA 32
- 2 **Architectural Overview** 33
  - 2.1 Introduction 33
  - 2.2 Memory Organization 33
    - 2.2.1 Register File 33
    - 2.2.2 Data Memory 34
    - 2.2.3 Code Memory 36
    - 2.2.4 Special Function Registers 37
  - 2.3 CPU 38
    - 2.3.1 CPU Blocks 39
  - 2.4 Task Management 43
  - 2.5 Instruction Set 44
    - 2.5.1 Instruction Syntax 44
    - 2.5.2 Instruction Set Summary 47
  - 2.6 External Bus 50
    - 2.6.1 External Bus Signals 50
    - 2.6.2 Bus Configuration 50
    - 2.6.3 Bus Timing 51
  - 2.7 Ports 52
  - 2.8 Peripherals 53
  - 2.9 80C51 Compatibility 53
    - 2.9.1 Software Compatibility 54
    - 2.9.2 Hardware Compatibility 54
- 3 **XA Memory Organization** 57
  - 3.1 Introduction 57
  - 3.2 The XA Register File 57
    - 3.2.1 Register File Overview 57
  - 3.3 The XA Memory Spaces 60
    - 3.3.1 Bytes, Words, and Alignment 61
  - 3.4 Data Memory 61
    - 3.4.1 Alignment in Data Memory 61
    - 3.4.2 External and Internal Overlap 61
    - 3.4.3 Use and Read/Write Access 62
    - 3.4.4 Data Memory Addressing 62
  - 3.5 Code Memory 65
    - 3.5.1 Alignment in Code Memory 66
    - 3.5.2 External and Internal Overlap 66
    - 3.5.3 Access 67
  - 3.6 Special Function Registers (SFRs) 67
  - 3.7 Summary of Bit Addressing 70
- 4 **CPU Organization** ....
  - 4.1 Introduction ....
  - 4.2 Program Status Word ....
    - 4.2.1 CPU Status Flags ....

CONTENTS

4.2.2 Operating Mode Flags 74
4.2.3 Program Writes to PSW 74
4.2.4 PSW Initialization 75
4.3 System Configuration Register 75
4.3.1 XA Large-Memory Model Description 76
4.3.2 XA Page 0 Model Description 76
4.4 Reset 77
4.4.1 Reset Sequence Overview 77
4.4.2 Power-up Reset 78
4.4.3 Internal Reset Sequence 78
4.4.4 XA Configuration at Reset 79
4.4.5 The Reset Exception Interrupt 80
4.4.6 Startup Code 81
4.4.7 Reset Interactions with XA Subsystems 81
4.4.8 An External Reset Circuit 81
4.5 Oscillator 82
4.6 Power Control 82
4.6.1 Idle Mode 83
4.6.2 Power-Down Mode 83
4.7 XA Stacks 84
4.7.1 The Stack Pointers 84
4.7.2 PUSH and POP 84
4.7.3 Stack-Based Addressing 86
4.7.4 Stack Errors 86
4.7.5 Stack Initialization 87
4.8 XA Interrupts 88
4.8.1 Interrupt Type Detailed Descriptions 89
4.8.2 Interrupt Service Data Elements 93
4.9 Trace Mode Debugging 95
4.9.1 Trace Mode Operation 96
4.9.2 Trace Mode Initialization and Deactivation 97
5 Real-time Multitasking 99
5.1 Assist for Multitasking in XA 99
5.1.1 Dual stack approach 99
5.1.2 Register Banks 100
5.1.3 Interrupt Latency and Overhead 100
5.1.4 Protection 100
6 Instruction Set and Addressing 103
6.1 Addressing Modes 103
6.2 Description of the Modes 104
6.2.1 Register Addressing 104
6.2.2 Indirect Addressing 105
6.2.3 Indirect-Offset Addressing 106
6.2.4 Direct Addressing 107
6.2.5 SFR Addressing 108
6.2.6 Intermediate Addressing 108
6.2.7 Bit Addressing 109
6.3 Relative Branching and Jumps 110
6.4 Data Types in XA 111
6.5 Instruction Set Overview 111
6.6 Summary of Illegal Operand Combinations on the XA 275
7 External Bus 277
7.1 External Bus Signals 277
7.1.1 PSEN – Program Store Enable 277
7.1.2 RD – Read 277
7.1.3 WRL – Write Low Byte 277
7.1.4 WRH – Write High Byte 277
7.1.5 ALE – Address Latch Enable 277
7.1.6 Address Lines 278
7.1.7 Multiplexed Address and Data Lines 278
7.1.8 WAIT – Wait 278
7.1.9 EA – External Access 278
7.1.10 BUSW – Bus Width 279

- 7.2 Bus Configuration ... 279
  - 7.2.1 8-Bit and 16-Bit Data Bus Widths ... 279
  - 7.2.2 Typical External Device Connections ... 281
- 7.3 Bus Timing and Sequences ... 283
  - 7.3.1 Code Memory ... 283
  - 7.3.2 Data Memory ... 285
  - 7.3.3 Reset Configuration ... 291
- 7.4 Ports ... 291
  - 7.4.1 I/O Port Access ... 291
  - 7.4.2 Port Output Configuration ... 292
  - 7.4.3 Quasi-Bidirectional Output ... 293
  - 7.4.4 Reset State and Initialization ... 296
  - 7.4.5 Sharing of I/O Ports with On-Chip Peripherals ... 296

8 Special Function Register Bus ... 297
- 8.1 Implementation and Possible Enhancements ... 297
- 8.2 Read-Modify-Write Lockout ... 298

9 80C51 Compatibility ... 299
- 9.1 Compatibility Considerations ... 299
  - 9.1.1 Memory Map and Addressing ... 299
  - 9.1.2 Interrupt and Exception Processing ... 301
  - 9.1.3 On-Chip Peripherals ... 302
  - 9.1.4 Bus Interface ... 302
  - 9.1.5 Instruction Set ... 303
- 9.2 Code Translation ... 306
- 9.3 New Instructions on the XA ... 309

Section 3 – XA Family Derivatives

XA-G1 CMOS single-chip 16-bit microcontroller ... 317
XA-G2 CMOS single-chip 16-bit microcontroller ... 346
XA-G3 CMOS single-chip 16-bit microcontroller ... 375

Section 4 – Future Derivatives

XA-C3 16-bit microcontroller with on-chip CAN ... 407
XA-S3 Single-chip 16-bit microcontroller ... 408

Section 5 – Application Notes

AN700 Digital filtering using XA Revision 0.11 ... 413
AN701 SP floating point math with XA ... 418
AN702 High level language support in XA ... 441
AN703 XA benchmark versus the architectures 68000, 80C196, and 80C51 ... 445
AN704 An upward migration path for the 80C51: the Philips XA architecture ... 469

Section 6 – Development Support Tools

XA tools linecard ... 478
Ashling: CTS51 Microprocessor development systems for Philips microcontrollers ... 479
CEIBO: DB-XA Development Board ... 484
EDI Corporation: Accessories for 8051-Architecture Devices ... 486
Emulation Technology, Inc.: XA Microcontroller Development Tools ... 487
Hi-Tech C Compiler for the Philips XA microcontroller – technical specifications ... 489
Logical Systems: Adapters for Philips 51XA-G3 ... 492
Nohau: EMUL 51XA In-Circuit Emulator for the Philips 80C51XA ... 493
Philips Semiconductors / Macraigor Systems: 80C51XA Software Development Tools ... 495
P51XA Development Board/Emulator ... 497
Signum Systems: Universal In-Circuit Emulator for 8051/31 Series ... 499
System General: Universal Device Programmers ... 501

Section 7 – Package Information

Soldering ... 505
SDIP42: plastic shrink dual in-line package; 42 leads (600 mil) ... SOT270-1 ... 507
LQFP44: plastic low profile quad flat package; 44 leads; body 10 x 10 x 1.4 mm ... SOT389-1 ... 508
PLCC44: plastic leaded chip carrier; 44 leads ... SOT187-2 ... 509
44-pin cerquad J-bend (K) package ... 1472A ... 510

Appendix A – Data Handbook System ... 512

Ordering Information

XA PRODUCTS

Example: P51XA G3 7 K B A

Philips 80C51 eXtended Architecture — P51XA

Derivative Name — G3

Package Code
A = Plastic Leaded Chip Carrier (PLCC)
B = Quad Flat Pack (QFP)
BD = Thin Quad Flat Pack (TQFP)
FA = Hermetic Cerdip (window)
KA = CerQuad (window)
N = Plastic Dual In-Line

Temperature
B = 0°C to +70°C
F = –40°C to +85°C
H = –40°C to +125°C

Speed
E = 16MHz
G = 20MHz
I = 24MHz
K = 30MHz

Memory Opation
0 = ROMless
3 = ROM
5 = Bond-Out (emulation)
7 = EPROM/OTP
9 = FEEPROM (FLASH)

Product Status

DEFINITIONS

Data Sheet Identification	Product Status	Definition
Objective Specification	**Formative or in Design**	This data sheet contains the design target or goal specifications for product development. Specifications may change in any manner without notice.
Preliminary Specification	**Preproduction Product**	This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Product Specification	**Full Production**	This data sheet contains Final Specifications. Philips Semiconductors reserves the right to make changes at any time without notice, in order to improve design and supply the best possible product.

XA tools linecard

	Telephone/Contact					Product
	North America		Europe			
C Compilers						
Archimedes	1–206–822–6300	Mary Sorensen	SW	41.61.331.7151	Claude Vonlanthen	C–51XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	C–XA
Hi–Tech	1–207–236–6713	Avocet – T. Taylor	UK	44.1.932.829460	Computer Solutions	Hi–Tech C (XA)
Franklin Software	1–408–296–8051	Siegfried Bleher	US	1–408–296–8051	Siegfried Bleher	XA–CD (5050)
Sierra Systems	1–510–339–1976	Larry Rosenthal	US	1.510.339.1976	Larry Rosenthal	Sierra C (XA)
Emulators (including Debuggers)						
Ashling	1–508–366–3220	Bob Labadini	IR	353.61.336644	Micheal Healy	Ultra2000–XA
Cactus Logic	1–818–337–4547	Joel Lagerquist	US	1.818.337.4547	Joel Lagerquist	IDS
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	DS–XA
Emulation Tech	1–408–982–0660	Joseph J. Bagliere	US	1.408.982.0660	Joseph J. Bagliere	Various
Nohau	1–408–866–1820	Steve Ehert	SW	46.40.922425	Mikael Johnsson	EMUL51XA–PC
Cross Assemblers						
Archimedes	1–206–822–6300	Mary Sorensen	SW	41.61.331.7151	Claude Vonlanthen	A–51XA
Ashling	1–508–366–3220	Bob Labadini	IR	353.61.336644	Micheal Healy	SDS–XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	ASM–XA
Franklin Software	1–408–296–8051	Siegfried Bleher	US	1–408–296–8051	Siegfried Bleher	XA–ASM (4050)
Philips/Macraigor*	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	Mcgtool
Real–Time Operating Systems						
CMX Company	1–508–872–7675	Charles Behrmann	US	1.508.872.7675	Charles Behrmann	CMX–RTX RTOS
Embedded Sys Prods	1–713–728–9688	Ron Hodge	US	1.713.728.9688	Ron Hodge	RTXC
R&D Publications	1–913–841–1631	Customer Service	US	1.913.841.1631	Customer Service	Labrosse MCU/OS
Simulators						
Archimedes	1–206–822–6300	Mary Sorensen	SW	41.61.331.7151	Claude Vonlanthen	SimCASE–51XA
Avocet Systems	1–207–236–9055	Jamie Arrison	US	1.207.236.9055	Jamie Arrison	AvCase–51XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	DEBUG–XA
Franklin Software	1–408–296–8051	Siegfried Bleher	US	1–408–296–8051	Siegfried Bleher	XA–DK (8250)
Philips/Macraigor*	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	Mcgtool
Translators (80C51–to–XA)						
Ashling	1–508–366–3220	Bob Labadini	IR	353.61.336644	Micheal Healy	N.A. — FC–51XA Eur. — Ultra2000–XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	CONV–XA
Philips/Macraigor*	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	Mcgtool
Development Boards						
Ceibo	1–314–830–4084	Roy Schwartzman	GE	49.6151.27505	M. Kimron	DB–XA
Philips/Macraigor	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	P51XA–DBE SD
EPROM Programmers						
BP Microsystems	1–800–225–2102	Sales Department	US	1.713.688.4600	Sales Department	BP–1200
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	MP–51
Data I/O Corp.	1–800–247–5700	Tech Help Desk	BE	32.1.638.0808	Roland Appeltants	UniSite
Philips/Macraigor	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	P51XA–DBE SD
Adapters & Sockets						
EDI Corp	1–702–735–4997	Milos Krejcik	US	1.702.735.4997	Milos Krejcik	44PG/44PL
Logical Systems	1–315–478–0722	Lynn Burko	US	1.315.478.0722	Lynn Burko	PA–XG3FC–44

* The Philips cross assembler, simulator, and translator are available on the Philips BBS. Call 1–408–991–2406 or 31–40–721102. File name XA–TOOLS.EXE

Microcontroller bulletin boards

To better serve our customers, Philips maintains two microcontroller bulletin boards. These computer bulletin board systems feature microcontroller newsletters, application and demonstration programs for download, and the ability to send messages to microcontroller application engineers.

The telephone numbers are:

North American Bulletin Board
300/1200/2400 baud 8-N-1
(800) 451-6644 (in the U.S.)
or
(408) 991-2406

European Bulletin Board
MAX 14.400 baud
Standards V32/V42/V42.bis/HST
+31 40 721102

European Application Help Desk
+31 40 722749
9a.m. – 16p.m. CET (Central European Time)

Sunnyvale ROMcode Bulletin Board

We also have a ROM code bulletin board through which you can submit ROM codes. This is a closed bulletin board for security reasons. To get an ID, contact your local sales office. The system can be accessed with a 2400, 1200, or 300 baud modem, and is available 24 hours a day.

The telephone number is:

(408) 991-3459

The following application note files are available on the Philips BBS:

App Note	BBS file name	App Note	BBS file name
AN417	PRN256K.ZIP	AN434	I2CPCKB.ZIP
AN420	INTRUPTS.ASM	AN435	IIC_OS.ZIP
AN422	I2CAPP.ZIP	AN438	I2C528.EXE
AN423	RS751.ASM	AN439	BATTCHRG.C
AN424	WARMBOOT.ZIP	AN440	BOOTSTRP.ZIP
AN425	I2C8584.ZIP	AN443	MAZEMOUS.ZIP
AN427	TIMERI.ZIP	AN445	ABMOUSE.ZIP
AN428	DEMO752.ASM	AN446	DUPUART.ZIP
AN429	AN429.ZIP	AN447	AUTOBAUD.ZIP
AN430	MM751.ZIP	EIE/AN91007	MM751B.ZIP
AN433	SLV751.ZIP	EIE/AN91009	EEPRM851.ZIP

Articles:

Add text overlay to any video display
CCI6.ZIP, MTV.ZIP

Philips Fax-On-Demand System

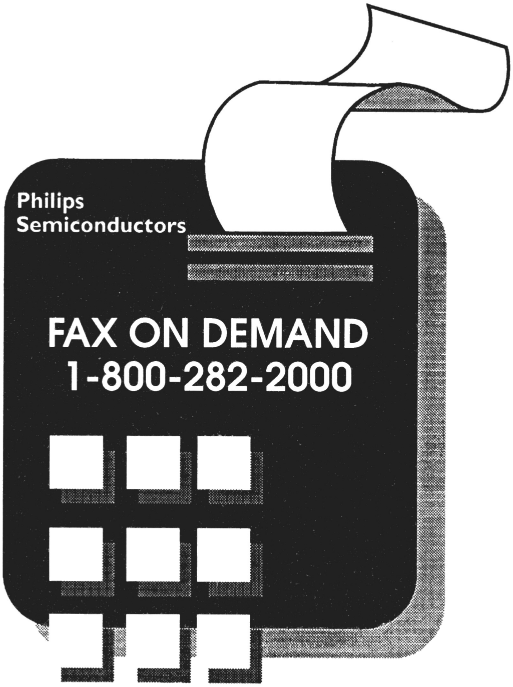
What's the number?

1-800-282-2000

What is it?

The Fax-On-Demand system is a computer facsimile system that allows customers to receive selected documents by Fax automatically.

Philips Fax-On-Demand system is now set up to fax selected datasheets as customers request them, 24 hours per day, 7 days per week.



How does it work?

Each time the system receives a call, the voice card plays the pre-recorded messages, and waits for the caller's responses. Based on the caller's responses, the system will find and fax the appropriate document to the caller's fax machine.

To receive a document, the user must know the document number. This number can be obtained by asking for an index of available documents the first time that he/she calls the system.

How is it set up?

The Philips Fax-On-Demand system has six indexes (so far):

1. Communication
2. Audio & Video (in progress)
3. Microcontrollers
4. Logic
5. Linear
6. PLD

So far, the system has 600 data sheets. We expect this number to rise to 800 data sheets very soon and keep increasing every quarter. As you know, it will take approximately one minute to fax one page. This wouldn't be that bad if the number of pages is less than ten. But if the document is 37 pages long, be ready for a long transmission!!!

Philips Fax-On-Demand number is 1-800-282-2000, try it!

The following listing of products are those for which documents are available via "Fax-On-Demand" for the communications category.

Coming soon: Fax-On-Demand for our European-produced products.

Who do I contact if I have any questions about Fax-On-Demand?

Hamid Mohammadi
Phone: (408) 991-4895
Technical Support Center

CMOS and NMOS 8-bit microcontroller family

80C51 FAMILY CMOS

TYPE	ROM/ EPROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PHILIPS PROBES	THIRD PARTY PODs
80C31 80C51 87C51	0 4k ROM 4k EPROM	128 128 128	33 33 33	DIL40, LCC44 QFP44	UART, 2 timers	87C51:QFP package up to 16MHz	OM1092 + OM1097 (16MHz) OM4120S	8052PC(M) POD-C51B(N)
83C51FA 87C51FA	8k ROM 8k EPROM	256 256	24 24	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA		PDS51FBSD	8351FX(M) POD-C51FX(N)
83L51FA 87L51FA	8k ROM 8k EPROM	256 256	20 20	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA	3V to 4.5V operation		POD-L51P(N)
87C51FB 83C51FB	16k ROM 16k EPROM	256 256	24 24	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA		PDS51FBSD	8351FX(M) POD-C51FX(N)
87L51FB 83L51FB	16k ROM 16k EPROM	256 256	20 20	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA	3V to 4.5V operation		POD-L51P(N)
87C51FC 83C51FC	32k ROM 32k EPROM	256 256	24 24	DIL40, LCC44 QFP44	Enhanced UART, 3 timers, PCA			8351FX(M) POD-C51FX(N)
80C32 80C52 87C52	0 8k ROM 8k EPROM	256 256 256	24 24 24	DIL40, LCC44 QFP44	UART, 3 timers		OM1079 OM5012	8052PC(M) POD-C32(N)
80C54 87C54	16k ROM 16k EPROM	256 256	24 24	DIL40, LCC44 QFP44	UART, 3 timers		OM1079 OM5012	8052PC(M) POD-C32(N)
80C58 87C58	32k ROM 32k EPROM	256 256	24 24	DIL40, LCC44 QFP44	UART, 3 timers		OM1079 OM5012	8052PC(M) POD-C32(N)
80C451 83C451 87C451	0 4k ROM 4k EPROM	128 128 128	16 16 16	DIP64/LCC68	UART, 2 timers Extended I/O		OM4123	83C451PC(M) POD-C451B(N)
83C504 87C504	16K ROM 16K EPROM	256 256	24 24	DIL40, LCC44 QFP44	24 by 8 divide, 2 timers			
87C524 83C524	16K EPROM 16k ROM	512 512	20 12	DIL40/LCC44 QFP44	UART, 3 timers Watchdog timer Bit I^2C		OM4111 + OM4110 + OM4120S	83528PC(M) POD-C528(N)
83C528 87C528 83CE528	32k ROM 32k EPROM 32kROM	512 512 512	16 16, 20 16	DIL40/LCC44 (QFP44) CE ONLY QFP	UART, 3 timers Watchdog timer Bit I^2C		OM4111 + OM4110 + OM4120S	83C528PC(M) POD-C528(N)
80C550 83C550 87C550	0 4k ROM 4k EPROM	128 128 128	16 16 16	LCC44 DIL40	UART, 2 timers 8 8-bit ADC inputs, watchdog timer		OM5055 + OM4110	83550(M) POD-C550(N)
80C552 83C552 87C552	0 8k ROM 8k EPROM	256 256 256	16, 24 16, 24 16	LCC68/QFP80	UART, 2 timers Timer with compare and capture, 2 PWM outputs, 8 10-bit ADC inputs, Byte I2C		OM1092 + OM1095 + OM4120S OM4128	83C552PC(M) POD–C552B(N)
83CE558 89CE558 80CE558	32K ROM 32K FLASH 0	1K 1K	16 16	QFP80	As 8xC552 with PLL-oscillator Auto scan ADC	89C: Q4-92 83C: Q2/3-93	OM4247	
80C562 83C562	0 8k ROM	256 256	16 16	LCC68/QFP80	UART, 2 timers Timer with compare and capture, 2 PWM outputs, 8 8-bit ADC inputs		OM1092 + OM1095 + OM4120S	83C552PC(M) POD-C552B(N)
80C575 83C575 87C575	0 8k 8k EPROM	256 256 256	16 16 16	DIL40, LCC44 QFP44	3 timers 1 Enh. UART, PCA, 4 analog comparators			POD-C575(N)
83C576 87C576	8k ROM 8k EPROM	256 256	16 16	DIL40, LCC44, SDIL42	10-bit A/D, 3 timers, PCA, Watchdog timer			
80C592 83C592 87C592	0 16k ROM 16k EPROM	512 512 512	16 16 16	LCC68/QFP80	8XC552 + CAN interface		OM4110 + OM4112 + OM4120S	POD-592(N)

M = Metlink
N = Nohau

80C51 FAMILY CMOS (Continued)

TYPE	ROM/ EPROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PHILIPS PROBES	THIRD PARTY PODs
87CE598 87CE598 80CE598	32K ROM 32K EPROM 0	512 512 512	16 16 16	QFP80	8xC552 + CAN interface. No I^2C	87CE: prod: Q2'94		
80C652 83C652 87C652	0 8k ROM 8k EPROM	256 256 256	16, 24 16, 24 16, 20	DIL40/LCC44 QFP44	UART, 2 timers Byte I^2C		OM1092 + OM1096 + OM4120S	83652PC(M) POD-C51B(N)
83C654 87C654	16k ROM 16k EPROM	256 256	16,24 16,20	DIL40/LCC44 QFP44	UART, 2 timers Byte I^2C		OM1092 + OM1096 + OM4120S	83654(M) POD-C51B(N)
83CE654 80CE654	16k ROM 0	256 256	16 16	QFP44	UART, 2 timers Byte I^2C	83C654 with Electromagnetic Compatibility improvements	OM1092 + OM1096 + OM4120S	83654(M) POD-C51B(N)
83C750 87C750	1K ROM 1KEPROM	64 64	40 40	SDIP24 skinny	1 timer		OM1094	83751PC(M)
83C751 83C748 87C751 87c748	2k ROM 2k EPROM	64 64	16 16	DIP24 skinny LCC28 DIP24 skinny	1 timer Bit I^2C (8XC751 only)		OM1094P	83751PC(M) POD-C751(N)
83C752 83C749 87C752 87c752	2k ROM 2k EPROM	64 64	16 16	DIP28, LCC28 DIP 28, LCC28	1 timer, PWM output, 5 8-bit ADC inputs, Bit I^2C (8XC752 only)		OM5072	83752A(M) POD-C752(N)
80C851 83C851	0 4k ROM	128 128	16 16	DIL40/LCC44 QFP44	UART, 2 timers 256 byte		OM1092 + OM4120S	80851PC(M) POD-C51(N)
83C852	6k ROM	256	6		2k byte EEPROM smart card hardware CU			
83C055 87C055	16k ROM 16k EPROM	256 256	12 12	DIP42 Shrunk DIP42 Shrunk	As 8XC053	In dev.	OM5054	

* The following microcontollers have no external memory access: 8XC751, 8XC752, 8XC053, 87C054, 83C852.
M = Metlink
N = Nohau

80CLXXX FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
85CL000	0	256	12	Piggyback	Piggyback CL410, CL411, CL51, P80C51			
85CL580	0	256	12	Piggyback	Piggyback CL580			
85CL781	0	256	12	Piggyback	Piggyback CL781, CL782, CL52			
80CL51 80CL31	4K 0	128 128	16 16	DIL40 VSO40	2 timers, UART		OM1079	QFP: OM5020
83CL410 80CL410	4k 0	128 128	12 12	DIL40 VSO40	2 timers Byte I^2C		OM1079	QFP: OM5020
83CL580	6k	256	16	QFP64/ VSO56	3timers, UART Watchdog timer Byte I^2C, 1 PWM 4*8 bit ADC		OM1079 + OM5004	OM1079: Probe base OM5004: Probe adap
83CL781 83CL782	16k 16k	256 256	12 @ 4.5V 12 @ 3V	DIL40 QFP44	3timers, UART Byte I^2C		OM1079 + OM5004 + tbd	OM1079: Probe base OM5004: Probe adap
83CL167 83CL267	16K 12K	256 256	12 12	SDIL64 QFP64	3timers 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 4*4 bit ADC Byte I^2C 160 char OSD 126 char fonts 4 char sizes Shadow modes ODS PLL osc. 10MHz Blinking	In Dev	OM4840 OM1079	
83CL168 83CL268	16K 12K	256 256	12 12	SDIL64 QFP64	3timers 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 4*4 bit ADC RC preprocessor Byte I^2C 3 wire serial I/O 160 char OSD 126 char fonts 4 char sizes Shadow modes ODS PLL osc. 10MHz Blinking	In Dev	OM4840 + OM1079	

8051 FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	THIRD PARTY EMULATOR
8051 8031	4k 0	128 128	15 15	DIL40/PLCC44 DIL40/PLCC44	UART, 2 timers		OM1092 + OM1097 + OM4120S	8052PC(M) OPD-C51B(N)
8052 8032	8k 0	256 256	15 15	DIL40/PLCC44 DIL40/PLCC44	UART, 3 timers UART, 3 timers		OM4111 + OM4110 + OM4120S	8052PC(M) OPD-C51B(N)

CMOS and NMOS 8-bit microcontroller family

8400 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
84C21A	2k	64	10	DIL28/SO28	20 I/O lines		OM1083	OM1025
84C41A	4k	128	10	DIL28/SO28	8-bit timer			(LSDS)
84C81A	8k	256	10	DIL28/SO28	Byte I^2C			
84C22A	2k	64	10	DIL20/SO20	13 I/O lines		OM1083 +	OM1025
84C42A	4k	64	10	DIL20/SO20	8-bit timer		Adapter_1	(LSDS)
84C12A	1k	64	16	DIL20/SO20				
				DIL20/SO20				
84C00B	0	256	10	28 pins	20 I/O lines	Piggyback	OM1080	
					8-bit timer			
					Byte I^2C			
84C00T	0	256	10	VSO-56		ROMless	OM1080	
84C121	1k	64	10	DIL20/SO20	13 I/O lines		OM1073	OM1025(LEDS)
					2 8-bit timers			
					8 bytes			
84C121B	0	64	10		EEPROM	Piggyback		OM1027
84C122A	1k	32	10	A: SO20	Controller for		OM4830	
84C122B				B: SO24	remote control			
84C422A	4K	32		C: SO28	A: 12 I/O			
84C422B					B: 16 I/O			
84C822A	8K	32			C: 20 I/O			
84C822B								
84C822C								
84C230	2l	64	10	DIL40/VSO40	12 I/O lines		OM1072	
					8-bit timer			
					16*4 LCD drive			
84C430	4k	128	10	QFP64	24 I/O lines		OM1072	
					8-bit timer			
					Byte I^2C			
					24*4 LCD drive			
84C430BH	0	128	10			Piggyback for C230 and C430		
84C633	6k	256	16	VSO56	28 I/O lines		OM1086	
					8-bit timer			
					16-bit up/down counter			
					16-bit timer			
					with compare			
					and capture			
84C633B	0	256	16		16*4 LCD drive			
84C440	4k	128	10	DIP42 shrunk	RC: 29 I/O lines	I^2C, RC	OM1074	For emulation of
84C441	4k	128	10		LC: 28 I/O lines	I^2C, LC		LC versions,
84C443	4k	128	10		8-bit timer	RC		use OM1074 +
84C444	4k	128	10		1 14-bit PWM	LC		adapter_3 +
84C640	6k	128	10		5 6-bit PWM	I^2C, RC		2 adapter_5
84C641	6k	128	10		3-bit ADC	I^2C, LC		
84C643	6k	128	10		OSD 2L-16	RC		
84C644	6k	128	10			LC		
84C840	8k	192	10			I^2C, RC		
84C841	8k	192	10			I^2C, LC		
84C843	8k	192	10			RC		Baud for LCDS
84C844	8k	192	10			LC		OM4831

8400 FAMILY CMOS (Continued)

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
84C646	6k	192	10	DIP42 shrunk	30 I/O lines DOS clock = PLL 8 bit timer 1-14 bit PWM 4-6 bit PWM 4-7 bit PWM 3-4 bit ADC DOS: 64 disp. RAM 62 char. fonts Char. blinking Shadow modes 8 foreground colors/char. 8 background colors/word DOS: clock: 8 . . 20MHz	I^2C, RC	OM4829 +	OM4833 for
84C846	8k	192	10			I^2C, RC	OM4832	LCD584
84C85	8k	256	10	DIL40/VSO40	32 I/O lines 8-bit timer Byte I^2C		OM1070	
84C85B	0	256	10			Piggyback for C85		
84C853	8k	256	16	DIL40/VSO40	33 I/O lines 8-bit timer 16-bit up/down counter 16-bit timer with compare and capture		OM1081	
84C853B	0	256	16			Piggyback for C853		
84C270	2k	128	10	DIL40/VSO40	8 I/O lines 16*8 capture keyboard matrix 8-bit timer		OM1077	
84C470	4k	128	10	DIL40/VSO40				
84C270B	0	128	10			Piggyback for C270		
84C470B	0	128	10		470 also handles mech. keys	Piggyback for C470		
84C271	2k	128	10	DIL40	8 I/O lines 16*8 mech. keyboard matrix 8-bit timer		OM1078	

8400 FAMILY NMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	EMULATOR TOOLS	REMARKS
8411	1k	64	6	DIL28/SO28	20 I/O lines 8-bit timer Byte I^2C			OM1025 (LCDS) + OM1026
8421	2k	64	6	DIL28/SO28				
8441	4k	128	6	DIL28/SO28				
8461	6k	128	6	DIL28/SO28				
8422	2k	64	6	DIL20	13 I/O lines 8-bit timer Bit I^2C			
8442	4k	128	6	DIL20				
8401B	0	128	6	28-pin		Piggyback for 84X1		

3300 FAMILY CMOS

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
3315A	1.5k	160	10	DIL28/SO28	20 I/O lines 8-bit timer $V_{DD} > 1.8V$		OM1083	OM1025(LCDS)
3343	3k	224	10	DIL28/SO28	20 I/O lines 8-bit timer $V_{DD} > 1.8V$ Byte I^2C		OM1083	OM1025(LCDS)
3344A	2k	224	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator		OM1071	OM1025(LCDS) + OM1028
3346A	4k	128	10	DIL28/SO28	20 I/O lines 8-bit timer Byte I^2C 256 bytes EEPROM $V_{DD} < 1.8V$		OM1076	
3347	1.5k	64	3.58	DIL20/SO20	12 I/O lines 8-bit timer DTMF generator		OM1071 + Adapter_2	OM1025(LCDS) + OM1028
3348A	8k	256	10	DIL28/SO28	20 I/O lines 8-bit timer Byte I^2C $V_{DD} < 1.8V$		OM1083	OM1025(LCDS)
3349A	4k	224	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator		OM1071	OM1025(LCDS) + OM1028
3350A	8k	128	3.58	VSO64	30 I/O lines 8-bit timer DTMF generator 256 bytes EEPROM			
3351A	2k	64	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator 128 bytes EEPROM		OM5000	
3352A	6k	128	3.58	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator 128 byte EEPROM		OM5000	
3353A	6k	128	16	DIL28/SO28	20 I/O lines 8-bit timer DTMF generator Ringer out 128 bytes EEPROM	March '92	OM5000	
3354A	8k	256	16	QFP64	36 I/O lines 8-bit timer DTMF generator Ringer out 256 bytes EEPROM	June '92	OM4829 + OM5003	OM4829: Probe base
8755A	0	128	16	DIL28/SO28	8k OTP 20 I/O lines 8-bit timer DTMF generator Melody output 128 bytes EEPROM	In Development		
3301B						Piggyback for 3315, 3343, 3348	OM1083	
3344B						Piggyback for 3344, 3347, 3349	OM1071	
3346B						Piggyback for 3346	OM1076	

3300 FAMILY CMOS (Continued)

TYPE	ROM	RAM	SPEED (MHz)	PACKAGE	FUNCTIONS	REMARKS	PROBE SDS	REMARKS
3350B						Piggyback for 3350A	OM4829+ OM5003	
3351B						Piggyback for 3351A, 3352A, 3353A	OM5000	
3354B						Piggyback for 3354A	OM4829+ OM5010	

CMOS 16-bit microcontroller family

16-BIT CONTROLLERS (68000 ARCHITECTURE)

TYPE	(EP)ROM	RAM	SPEED (MHz)	FUNCTIONS	REMARKS	PHILIPS TOOLS	THIRD-PARTY TOOLS
68070	–	–	17.5	2 DMA channels, MMU, UART, 16-bit timer, I^{2}C, 68000 bus interface, 16Mb address range		OM4160 Microcore 1 OM4160/2 Microcore 2 OM4161 (SBE68070) OM4767/2 XRAY68070SBE high level symbolic debugger OM4222 68070DS development system OM4226 XRAY68070DS high level symbolic debugger	TRACE32-ICE68070 (Lauterbach)
93C101	34k	512	15	Derivative with low power modes	Not for new design		
90CE201	16MB external ROM	16MB external RAM	24	UART, fast I^{2}C, 3 timers (16 bit), Watchdog timer. 68000 software compatible, EMC, QFP64	–25 to +85°C	OM4162 Microcore 4	TRACE32 – (Lauterbach)

16-BIT CONTROLLERS (XA ARCHITECTURE)

TYPE	(EP)ROM	RAM	SPEED (MHz)	FUNCTIONS	REMARKS	DEVELOPMENT TOOLS
XA-G1	8k	512	30	3 timers, watchdog, 2 UARTs	–40 to +125°C	Nohau Ceibo MacCraiger Systems
XA-G2	16k	512	30	3 timers, watchdog, 2 UARTs	–40 to +125°C	Nohau Ceibo MacCraiger Systems
XA-G3	32k	512	30	3 timers, watchdog, 2 UARTs	–40 to +125°C	Nohau Ceibo MacCraiger Systems

80C51 microcontroller family features guide

Part Number (ROMless)		Memory			Counter Timers	I/O Port	Serial Interfaces	External Interrupt	Comments/ Special Features
		ROM	EPRM	RAM					
P	83C750	1K		64	1 (16-bit)	2-3/8	-	2	40 MHz, Lowest cost, SSOP
P	87C750		1K	64	1 (16-bit)	2-3/8	-	2	40 MHz, Lowest cost, SSOP
P	83C748	2K		64	1 (16-bit)	2-3/8	-	2	8XC751 w/o I^2C, SSOP
P	87C748		2K	64	1 (16-bit)	2-3/8	-	2	8XC751 w/o I^2C, SSOP
S	83C751	2K		64	1 (16-bit)	2-3/8	I^2C (bit)	2	24-pin Skinny DIP, SSOP
S	87C751		2K	64	1 (16-bit)	2-3/8	I^2C (bit)	2	24-pin Skinny DIP, SSOP
P	83C749	2K		64	1 (16-bit)	2-5/8	-	2	8XC752 w/o I^2C, SSOP
P	87C749		2K	64	1 (16-bit)	2-5/8	-	2	8XC752 w/o I^2C, SSOP
S	83C752	2K		64	1 (16-bit)	2-5/8	I^2C (bit)	2	5 Channel 8-bit A/D, PWM Output, SSOP
S	87C752		2K	64	1 (16-bit)	2-5/8	I^2C (bit)	2	5 Channel 8-bit A/D, PWM Output, SSOP
MAx	8051AH (8031AH)	4K		128	2	4	UART	2	NMOS
SC	80C51 (80C31)	4K		128	2	4	UART	2	CMOS (Sunnyvale)
PCx	80C51 (80C31)	4K		128	2	4	UART	2	CMOS (Hamburg)
SC	87C51		4K	128	2	4	UART	2	CMOS
P	80CL51 (80CL31)	4K		128	2	4	UART	10	Low Voltage (1.8V to 6V), Low Power
P	83CL410 (80CL410)	4K		128	2	4	I^2C	10	Low Voltage (1.8V to 6V), Low Power
SC	83C451 (80C451)	4K		128	2	7	UART	2	Extended I/O, Processor Bus Interface
SC	87C451		4K	128	2	7	UART	2	Extended I/O, Processor Bus Interface
P	83C550 (80C550)	4K		128	2 + Watchdog	4	UART	2	8 Channel 8-bit A/D
P	87C550		4K	128	2 + Watchdog	4	UART	2	8 Channel 8-bit A/D
P	83C851 (80C851)	4K		128	2	4	UART	2	256B EEPROM, 80C51 Pin compatible
P	83C542	4K		256	2	1	I^2C	2	ACCESS.bus, replaces 8042 KB controller
P	87C542		4K	256	2	1	I^2C	2	See Above
P	83C852	6K		256	2 (16-bit)	2/8	-	1	Smartcard Controller with 2K EEPROM (Data, Code) Cryptographic Calc Unit
P	83CL580 (80CL580)	6K		256	3 + Watchdog	5	UART, I^2C	9	4 Channel 8-bit A/D, PWM Output, Low Voltage (2.5V to 6V), Low Power
MAx	8052AH (8032AH)	8K		256	3	4	UART	2	NMOS
P	80C52 (80C32)	8K		256	3	4	UART	2	80C51 Pin Compatible
P	87C52		8K	256	3	4	UART	2	(see above)
P	83C652 (80C652)	8K		256	2	4	UART, I^2C	2	80C51 Pin Compatible
S	87C652		8K	256	2	4	UART, I^2C	2	(see above)
P	83C453 (80C453)	8K		256	2	7	UART	2	Extended I/O, Processor Bus Interface
P	87C453		8K	256	2	7	UART	2	Extended I/O, Processor Bus Interface
S	83C51FA (80C51FA)	8K		256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	87C51FA		8K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	83L51FA	8K		256	3 + PCA	4	UART	2	Low Voltage 83C51FA (3V @ 20MHz)
S	87L51FA		8K	256	3 + PCA	4	UART	2	Low Voltage OTP 87C51FA (3V @ 20MHz)
P	83C575 (80C575)	8K		256	3 + PCA+ Watchdog	4	UART	2	High Reliability, with Low Voltage Detect, OSC Fail Detect, Analog Comparators, PCA
P	87C575		8K	256	(see above)	4	UART	2	(see above)
P	83C576 (80C576)	8K		256	3 + PCA+ Watchdog	4	UART	2	Same as 8XC575 plus UPI and 10-bit A/D
P	87C576		8K	256	(see above)	4	UART	2	(see above)
PC	83C562 (80C562)	8K		256	3 + Watchdog	6	UART	2	8 Channel 8-bit A/D, 2 PWM Outputs, Capture/Compare Timer
PCx	83C552 (80C552)	8K		256	3 + Watchdog	6	UART, I^2C	2	8 Channel 10-bit A/D, 2 PWM Outputs, Capture/Compare Timer
S	87C552		8K	256	3 + Watchdog	6	UART, I^2C	2	(see above)

Notes: Part number prefixes are noted in the first column.
All combinations of part type, speed, temperature and package may not be available.

80C51 microcontroller family features guide

Part Number (ROMless)		Program Security?	Clock Freq (MHz)	Temperature Ranges (°C)			Package				
				0 to 70	–40 to +85	–55 to +125	PDIP	CDIP	PLCC	CLCC	PQFP/SSOP
83C750	S	N	3.5 to 40	X	X		N24	F24	A28		DB24 (0–70F)
87C750	S	Y	3.5 to 40	X	X		N24	F24	A28		DB24 (0–70F)
83C748	S	N	3.5 to16	X	X		N24		A28		DB24 (0–70F)
87C748	S	Y	3.5 to16	X	X		N24	F24	A28		DB24 (0–70F)
83C751	S	N	3.5 to16	X	X		N24		A28		DB24 (0–70F)
87C751	S	Y	3.5 to16	X	X		N24	F24	A28		DB24 (0–70F)
83C749	S	N	3.5 to 16	X	X		N28		A28		DB28 (0–70F)
87C749	S	Y	3.5 to 16	X	X		N28	F28	A28		DB28 (0–70F)
83C752	S	N	3.5 to 16	X	X	X	N28		A28		DB28 (0–70F)
87C752	S	Y	3.5 to 16	X	X	X	N28	F28	A28		DB28 (0–70F)
8051AH (8031AH)	S	N	3.5 to 15	X	X		N40		A44		
SC80C51 (80C31)	S	Y	3.5 to 33	X	X	X	N40		A44		B44 (5)
PCx80C51 (80C31)	H	N	1.2 to 30	X	X	X	P (40)		WP (44)		H (44)
87C51	S	Y	3.5 to 33	X	X	X	N40	F40	A44	K44	B44 (5)
80CL51 (80CL31)	Z	N	0 to 16 (1)		X		N40 (2)				B44
83CL410(80CL410)	Z	N	0 to 12 (1)		X		N40 (2)				B44
83C451 (80C451)	S	N	3.5 to 16	X	X	X	N64 (4)		A68		
87C451	S	Y	3.5 to16	X	X	X	N64 (4)		A68		
83C550 (80C550)	S	Y	3.5 to 16	X	X		N40		A44		
87C550	S	Y	3.5 to 16	X	X	–40 to +125	N40	F40	A44	K44	
83C851 (80C851)	H	Y	1.2 to 16	X	X		N40		A44		B44
83C542	S	Y	3.5 to 16	X					A44		
87C542	S	Y	3.5 to 16	X					A44	K44	
83C852	H	Y	1 to 12	X			SO28 or die				
83CL580 (80CL580)	Z	N	0 to 12 (1)		X		(3)				B64
8052AH (8032AH)	S	N	3.5 to 15	X	X		N40		A44		
80C52 (80C32)	S	Y	3.5 to 24	X	X		N40		A44		B44 (5)
87C52	S	Y	3.5 to 24	X	X	X	N40	F40	A44	K44	B44 (5)
83C652 (80C652)	H	Y	1.2 to 24	X	X	–40 to +125	N40		A44		B44
87C652	S	Y	1.2 to 20	X	X	X	N40	F40	A44	K44	
83C453 (80C453)	S	N	3.5 to 16	X	X				A68		
87C453	S	Y	3.5 to16	X	X				A68		
83C51FA (80C51FA)	S	Y	3.5 to 24	X	X		N40		A44		B44
87C51FA	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83L51FA	S	Y	3.5 to 20	X	X		N40		A44		B44
87L51FA	S	Y	3.5 to 20	X	X		N40	F40	A44	K44	B44
83C575 (80C575)	S	Y	4 to 16	X		X	N40		A44		B44
87C575	S	Y	4 to 16	X		X	N40	F40	A44	K44	B44
83C576 (80C576)	S	Y	4 to 16	X		X	N40		A44		B44
87C576	S	Y	4 to 16	X		X	N40	F40	A44	K44	B44
83C562 (80C562)	H	N	1.2 to 16	X	X	–40 to +125			A68		B80
83C552 (80C552)	H	N	1.2 to 30	X	X	–40 to +125			A68		B80
87C552	S	Y	1.2 to 16	X					A68	K68	

Notes: Production Centers are indicated in the second column: H – Hamburg, S – Sunnyvale, Z – Zurich.
All combinations of part type, speed, temperature and package may not be available.
1) Oscillator options start from 32kHz.
2) Also available in VSO40 package.
3) Also available in VSO56 Package.
4) Not recommended for new design.
5) Package available up to 16 MHz only.

80C51 microcontroller family features guide

Part Number (ROMless)		Memory			Counter Timers	I/O Port	Serial Interfaces	External Interrupt	Comments/ Special Features
		ROM	EPRM	RAM					
P	83CL267	12K		256	3	2 5/8	I^{2}C	–	OSD, 8 PWM Outputs, 3 Software A/D Inputs, 8 LED Drivers
P	83CL268	12K		256	3	2 5/8	I^{2}C, 1M Baud	–	(see above)
P	83C055	16K		256	2 (16-bit)	3 1/2	–	2	On-Screen Display, 9 PWM Outputs, 3 Software A/D Inputs
P	87C055		16K	256	2 (16-bit)	3 1/2	–	2	(see above)
P	80C54	16K		256	3	4	UART	2	Standard; 80C51 compatible
P	87C54		16K	256	3	4	UART	2	Standard; 87C51 compatible
P	83C504 (80C504)	16K		256	2	4	UART	2	'654 with Hardware Divide (no I^{2}C)
P	87C504		16K	256	2	4	UART	2	(see above)
P	83C654	16K		256	2	4	UART, I^{2}C	2	80C51 Pin Compatible
S	87C654		16K	256	2	4	UART, I^{2}C	2	(see above)
P	83CE654	16K		256	2	4	UART, I^{2}C	2	83C654 with Reduced EMI
P	83CL781	16K		256	3	4	UART, I^{2}C	10	Low Voltage (1.8V to 6V), Low Power
P	83CL782	16K		256	3	4	UART, I^{2}C	10	83CL781 Optimized 12MHz @ 3.1V
S	83C51FB	16K		256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	87C51FB		16K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	83L51FB	16K		256	3 + PCA	4	UART	2	Low Voltage 83C51FB (3V @ 20MHz)
S	87L51FB		16K	256	3 + PCA	4	UART	2	Low Voltage OTP 87C51FB (3V @ 20MHz)
P	83CL167	16K		256	3	6 1/8	I^{2}C	–	OSD, 8 PWM Outputs, 4 Software A/D Inputs, 8 LED Drivers
P	83CL168	16K		256	3	6 1/8	I^{2}C, 1M Baud	–	(see above)
P	83C524	16K		512	3 + Watchdog	4	UART, I^{2}C-bit	2	512 RAM
P	87C524		16K	512	3 + Watchdog	4	UART, I^{2}C-bit	2	512 RAM
P	83C592 (80C592)	16K		512	3 + Watchdog	6	UART, CAN	6	CAN Bus Controller with 8 x 10-bit A/D, 2 PWM outputs, Capture/Compare Timer
P	87C592		16K	512	3 + Watchdog	6	UART, CAN	6	(see above)
P	80C58	32K		256	3	4	UART	2	Standard; 80C51 compatible
P	87C58		32K	256	3	4	UART	2	Standard; 87C51 compatible
S	83C51FC	32K		256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
S	87C51FC		32K	256	3 + PCA	4	UART	2	Enhanced UART, 3 timers + PCA
P	83C528 (80C528)	32K		512	3 + Watchdog	4	UART, I^{2}C-bit	2	Large Memory for High Level Languages
P	87C528		32K	512	3 + Watchdog	4	UART, I^{2}C-bit	2	Large Memory for High Level Languages
P	83CE528 (80CE528)	32K		512	3 + Watchdog	4	UART, I^{2}C-bit	2	8XC528 with Reduced EMI
P	83CE598 (80CE598)	32K		512	3 + Watchdog	6	UART, CAN	6	CAN Bus Controller, 8 x 10-bit A/D, 2 PWM outputs, WD, T2, Reduced EMI
P	87CE598		32K	512	3 + Watchdog	6	UART, CAN	6	(see above)
P	83CE558(80CE558)	32K		1024	3 + Watchdog	6	UART, I^{2}C	2	Low EMI, 8 Channel 10-bit A/D, 2 PWM Outputs, Capture/Compare Timer
P	89CE558		32K	1024	3 + Watchdog	6	UART, I^{2}C	2	32K FLash EEPROM plus above

Notes: Part number prefixes are noted in the first column.
All combinations of part type, speed, temperature and package may not be available.

80C51 microcontroller family features guide

Part Number (ROMless)		Program Security?	Clock Freq (MHz)	Temperature Ranges (°C)			Package				
				0 to 70	–40 to +85	–55 to +125	PDIP	CDIP	PLCC	CLCC	PQFP/SSOP
83CL267	T	N	4.0 to 12	X			R42				B64
83CL268	T	N	4.0 to 12	X			R42				B64
83C055	S	N	3.5 to 20	X			NB42				
87C055	S	N	3.5 to 20	X			NB42				
80C54	S	Y	3.5 to 24	X	X		N40		A44		B44
87C54	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83C504 (80C504)	S	Y	1.2 to 20	X	X	X	N40		A44		B44
87C504	S	Y	1.2 to 20	X	X	X	N40	F40	A44	K44	B44
83C654 (80C654)	H	Y	1.2 to 24	X	X	–40 to +125	R42, N40		A44		B44
87C654	S	Y	1.2 to 20	X	X	X	N40	F40	A44	K44	B44
83CE654	H	Y	1.2 to 16	X	X						B44
83CL781	Z	N	0 to 12 (1)		X		N40				B44
83CL782	Z	N	0 to 12 (1)		–25 to +55		N40				B44
83C51FB	S	Y	3.5 to 24	X	X		N40		A44		B44
87C51FB	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83L51FB	S	Y	3.5 to 20	X			N40		A44		B44
87L51FB	S	Y	3.5 to 20	X			N40	F40	A44	K44	B44
83CL167	T	N	4.0 to 12	X			R42				B64
83CL168	T	N	4.0 to 12	X			R42				B64
83C524	H	Y	1.2 to 16	X	X		N40		A44		B44
87C524	S	Y	3.5 to 20	X	X		N40	F40	A44	K44	B44
83C592 (80C592)	H	Y	1.2 to 16		X	–40 to +125			A68	K68	
87C592	H	Y	1.2 to 16	X			R42		A68	K68	
80C58	S	Y	3.5 to 16	X	X		N40		A44		B44
87C58	S	Y	3.5 to 16	X	X		N40	F40	A44	K44	B44
83C51FC	S	Y	3.5 to 24	X	X		N40		A44		B44
87C51FC	S	Y	3.5 to 24	X	X		N40	F40	A44	K44	B44
83C528 (80C528)	H	Y	1.2 to 16	X	X	–40 to +125	N40		A44		B44
87C528	S	Y	3.5 to 20	X	X		N40	F40	A44	K44	B44
83CE528 (80CE528)	H	Y	1.2 to 16	X	X	–40 to +125			A44		B44
83CE598 (80CE598)	H	Y	1.2 to 16		X	–40 to +125					B80
87CE598	H	Y	3.5 to 16	X	X						B80
83CE558 80CE558	H	Y	1.2 to 16	X	X	–40 to +125					B80
89CE558	H	Y	1.2 to 16	X	X					Q80	B80

Notes: Production Centers are indicated in the second column: H – Hamburg, S – Sunnyvale, Z – Zurich.
All combinations of part type, speed, temperature and package may not be available.
1) Oscillator options start from 32kHz.
2) Also available in VSO40 package.
3) Also available in VSO56 Package.
4) Not recommended for new design.
5) Package available up to 16 MHz only.

Handling MOS devices

ELECTROSTATIC CHARGES

Electrostatic charges can exist in many things; for example, man-made-fibre clothing, moving machinery, objects with air blowing across them, plastic storage bins, sheets of paper stored in plastic envelopes, paper from electrostatic copying machines, and people. The charges are caused by friction between two surfaces, at least one of which is non-conductive. The magnitude and polarity of the charges depend on the different affinities for electrons of the two materials rubbing together, the friction force and the humidity of the surrounding air.

Electrostatic discharge is the transfer of an electrostatic charge between bodies at different potentials and occurs with direct contact or when induced by an electrostatic field. All of our MOS devices are internally protected against electrostatic discharge but they **can** be damaged if the following precautions are not taken.

WORK STATION

Figure 1 shows a working area suitable for safely handling electrostatic sensitive devices. It has a work bench, the surface of which is conductive or covered by an antistatic sheet. Typical resistivity for the bench surface is between 1 and 500 kΩ per cm^2. The floor should also be covered with antistatic material. The following precautions should be observed:

- Persons at a work bench should be earthed via a wrist strap and a resistor
- All mains-powered electrical equipment should be connected via an earth leakage switch
- Equipment cases should be earthed
- Relative humidity should be maintained between 50 and 65%
- An ionizer should be used to neutralize objects with immobile static charges

RECEIPT AND STORAGE

MOS devices are packed for dispatch in antistatic/conductive containers, usually boxes, tubes or blister tape. The fact that the contents are sensitive to electrostatic discharge is shown by warning labels on both primary and secondary packing.

The devices should be kept in their original packing whilst in storage. If a bulk container is partially unpacked, the unpacking should be performed at a protected work station. Any MOS devices that are stored temporarily should be packed in conductive or antistatic packing or carriers.

ASSEMBLY

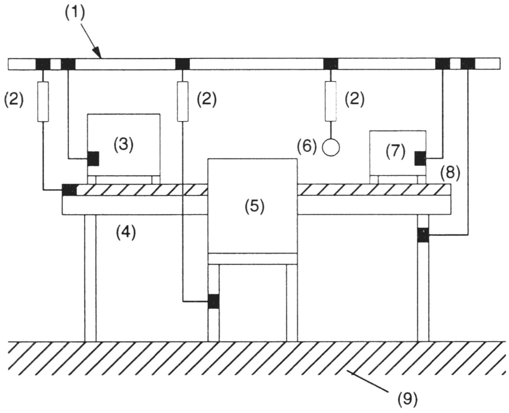
MOS devices must be removed from their protective packing with earthed component pincers or short-circuit clips. Short-circuit clips must remain in place during mounting, soldering and cleansing/drying processes. Do not remove more devices from the storage packing than are needed at any one time. Production/assembly documents should state that the product contains electrostatic sensitive devices and that special precautions need to be taken.

During assembly, endure that the MOS devices are the last of the components to be mounted and that this is done at a protected work station.

All tools used during assembly, including soldering tools and solder baths, must be earthed. All hand tools should be of conductive or antistatic material and, where possible, should not be insulated.

Measuring and testing of completed circuit boards must be done at a protected work station. Place the soldered side of the circuit board on conductive or antistatic foam and remove the short-circuit clips. Remove the circuit board from the foam, holding the board only at the edges. Make sure the circuit board does not touch the conductive surface of the work bench. After testing, replace the circuit board on the conductive foam to await packing.

Assembled circuit boards containing MOS devices should be handled in the same way a unmounted MOS devices. they should also carry waning labels and be packed in conductive or antistatic packing.



(1) Earthing rail.
(2) Resistor (500 kΩ ± 10%, 0.5 W).
(3) Ionizer.
(4) Work bench.
(5) Chair
(6) Wrist strap.
(7) Electrical equipment.
(8), (9) Conductive surface/antistatic sheet.

Figure 1. Protected work station

Section 2

XA User Guide

CONTENTS

1 The XA Family – High Performance, Enhanced Architecture 80C51-Compatible 16-Bit CMOS Microcontrollers 31
- 1.1 Introduction 31
- 1.2 Architectural Features of XA 32

2 Architectural Overview 33
- 2.1 Introduction 33
- 2.2 Memory Organization 33
  - 2.2.1 Register File 33
  - 2.2.2 Data Memory 34
  - 2.2.3 Code Memory 36
  - 2.2.4 Special Function Registers 37
- 2.3 CPU 38
  - 2.3.1 CPU Blocks 39
- 2.4 Task Management 43
- 2.5 Instruction Set 44
  - 2.5.1 Instruction Syntax 44
  - 2.5.2 Instruction Set Summary 47
- 2.6 External Bus 50
  - 2.6.1 External Bus Signals 50
  - 2.6.2 Bus Configuration 50
  - 2.6.3 Bus Timing 51
- 2.7 Ports 52
- 2.8 Peripherals 53
- 2.9 80C51 Compatibility 53
  - 2.9.1 Software Compatibility 54
  - 2.9.2 Hardware Compatibility 54

3 XA Memory Organization 57
- 3.1 Introduction 57
- 3.2 The XA Register File 57
  - 3.2.1 Register File Overview 57
- 3.3 The XA Memory Spaces 60
  - 3.3.1 Bytes, Words, and Alignment 61
- 3.4 Data Memory 61
  - 3.4.1 Alignment in Data Memory 61
  - 3.4.2 External and Internal Overlap 61
  - 3.4.3 Use and Read/Write Access 62
  - 3.4.4 Data Memory Addressing 62
- 3.5 Code Memory 65
  - 3.5.1 Alignment in Code Memory 66
  - 3.5.2 External and Internal Overlap 66
  - 3.5.3 Access 67
- 3.6 Special Function Registers (SFRs) 67
- 3.7 Summary of Bit Addressing 70

4 CPU Organization 71
- 4.1 Introduction 71
- 4.2 Program Status Word 72
  - 4.2.1 CPU Status Flags 72
  - 4.2.2 Operating Mode Flags 74
  - 4.2.3 Program Writes to PSW 74
  - 4.2.4 PSW Initialization 75
- 4.3 System Configuration Register 75
  - 4.3.1 XA Large-Memory Model Description 76
  - 4.3.2 XA Page 0 Model Description 76
- 4.4 Reset 77
  - 4.4.1 Reset Sequence Overview 77
  - 4.4.2 Power-up Reset 78
  - 4.4.3 Internal Reset Sequence 78
  - 4.4.4 XA Configuration at Reset 79
  - 4.4.5 The Reset Exception Interrupt 80
  - 4.4.6 Startup Code 81
  - 4.4.7 Reset Interactions with XA Subsystems 81
  - 4.4.8 An External Reset Circuit 81

- 4.5 Oscillator 82
- 4.6 Power Control 82
  - 4.6.1 Idle Mode 83
  - 4.6.2 Power-Down Mode 83
- 4.7 XA Stacks 84
  - 4.7.1 The Stack Pointers 84
  - 4.7.2 PUSH and POP 84
  - 4.7.3 Stack-Based Addressing 86
  - 4.7.4 Stack Errors 86
  - 4.7.5 Stack Initialization 87
- 4.8 XA Interrupts 88
  - 4.8.1 Interrupt Type Detailed Descriptions 89
  - 4.8.2 Interrupt Service Data Elements 93
- 4.9 Trace Mode Debugging 95
  - 4.9.1 Trace Mode Operation 96
  - 4.9.2 Trace Mode Initialization and Deactivation 97

5 Real-time Multitasking 99
- 5.1 Assist for Multitasking in XA 99
  - 5.1.1 Dual stack approach 99
  - 5.1.2 Register Banks 100
  - 5.1.3 Interrupt Latency and Overhead 100
  - 5.1.4 Protection 100

6 Instruction Set and Addressing 103
- 6.1 Addressing Modes 103
- 6.2 Description of the Modes 104
  - 6.2.1 Register Addressing 104
  - 6.2.2 Indirect Addressing 105
  - 6.2.3 Indirect-Offset Addressing 106
  - 6.2.4 Direct Addressing 107
  - 6.2.5 SFR Addressing 108
  - 6.2.6 Intermediate Addressing 108
  - 6.2.7 Bit Addressing 109
- 6.3 Relative Branching and Jumps 110
- 6.4 Data Types in XA 111
- 6.5 Instruction Set Overview 111
- 6.6 Summary of Illegal Operand Combinations on the XA 275

7 External Bus 277
- 7.1 External Bus Signals 277
  - 7.1.1 PSEN – Program Store Enable 277
  - 7.1.2 RD – Read 277
  - 7.1.3 WRL – Write Low Byte 277
  - 7.1.4 WRH – Write High Byte 277
  - 7.1.5 ALE – Address Latch Enable 277
  - 7.1.6 Address Lines 278
  - 7.1.7 Multiplexed Address and Data Lines 278
  - 7.1.8 WAIT – Wait 278
  - 7.1.9 EA – External Access 278
  - 7.1.10 BUSW – Bus Width 279
- 7.2 Bus Configuration 279
  - 7.2.1 8-Bit and 16-Bit Data Bus Widths 279
  - 7.2.2 Typical External Device Connections 281
- 7.3 Bus Timing and Sequences 283
  - 7.3.1 Code Memory 283
  - 7.3.2 Data Memory 285
  - 7.3.3 Reset Configuration 291
- 7.4 Ports 291
  - 7.4.1 I/O Port Access 291
  - 7.4.2 Port Output Configuration 292
  - 7.4.3 Quasi-Bidirectional Output 293
  - 7.4.4 Reset State and Initialization 296
  - 7.4.5 Sharing of I/O Ports with On-Chip Peripherals 296

8 Special Function Register Bus 297
- 8.1 Implementation and Possible Enhancements 297
- 8.2 Read-Modify-Write Lockout 298

9 80C51 Compatibility 299
- 9.1 Compatibility Considerations 299
  - 9.1.1 Memory Map and Addressing 299
  - 9.1.2 Interrupt and Exception Processing 301
  - 9.1.3 On-Chip Peripherals 302
  - 9.1.4 Bus Interface 302
  - 9.1.5 Instruction Set 303
- 9.2 Code Translation 306
- 9.3 New Instructions on the XA 309

1 The XA Family - High Performance, Enhanced Architecture 80C51-Compatible 16-Bit CMOS Microcontrollers

1.1 Introduction

The role of the microcontroller is becoming increasingly important in the world of electronics as systems which in the past relied on mechanical or simple analog electrical control systems have microcontrollers embedded in them that dramatically improve functionality and reliability, while reducing size and cost. Microcontrollers also provide the general purpose solutions needed so that common software and hardware can be shared among multiple designs to reduce overall design-in time and costs.

The requirements of systems using microcontrollers are also much more demanding now than a few years ago. Whether called by the name "microcontrollers", "embedded controllers" or "single-chip microcomputers", the systems that use these devices require a much higher level of performance and on-chip integration.

As microcontrollers begin to enter into more complex control environments, the demand for increased throughput, increased addressing capability, and higher level of on-chip integration has led to the development of 16-bit microcontrollers that are capable of processing much more information than 8-bit microcontrollers. However, simply integrating more bits or more peripheral functions does not solve the demand of the control systems being developed today. New microcontrollers must provide high-level-language support, powerful debugging environments, and advanced methods of real time control in order to meet the more stringent functionality and cost requirements of these systems.

To meet the above goals The XA or "eXtended Architecture" family of general-purpose microcontrollers from Philips is being introduced to provide the highest performance/cost ratio for a variety of high performance embedded-systems-control applications including real-time, multi-tasking environments. The XA family members add to the CPU core a specific complement of on-chip memory, I/Os, and peripherals aimed at meeting the requirements of different application areas. The core-based architecture allows easy expansion of the family according to a wide variety of customer requirements. The powerful instruction set supports faster computing power, faster data transfer, multi-tasking, improved response to external events and efficient high-level language programming.

Upward (assembly-level) code compatibility with the Philips 80C51 family of controllers provides a smooth design transition for system upgrades by providing tremendously enhanced performance.

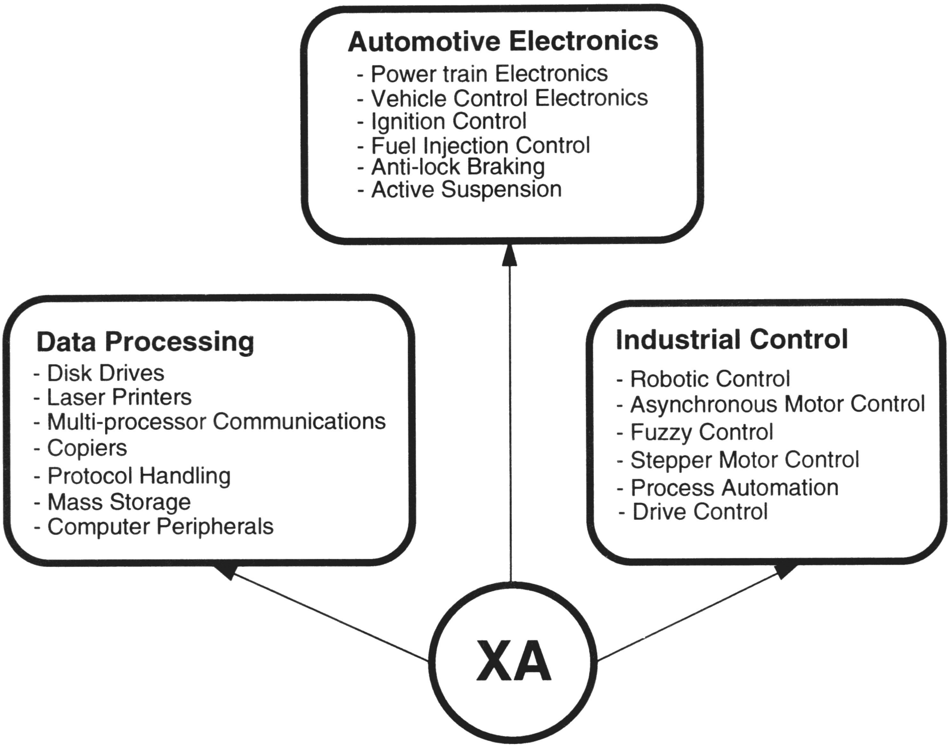

Automotive Electronics
- Power train Electronics
- Vehicle Control Electronics
- Ignition Control
- Fuel Injection Control
- Anti-lock Braking
- Active Suspension
Data Processing
- Disk Drives
- Laser Printers
- Multi-processor Communications
- Copiers
- Protocol Handling
- Mass Storage
- Computer Peripherals
Industrial Control
- Robotic Control
- Asynchronous Motor Control
- Fuzzy Control
- Stepper Motor Control
- Process Automation
- Drive Control
XA

Figure 1. Applications of Philips XA microcontrollers

1.2 Architectural Features of XA

- Upward compatibility with the standard 8XC51 core (assembly source level)
- 24-bit address range (16 Megabytes code and data space)
- 16-bit static CPU
- Enhanced architecture using both 16-bit words and 8-bit bytes
- Enhanced instruction set
- High code efficiency; most of the instructions are 2-4 bytes in length
- Fast 16X16 Multiply and 32x16 Divide Instructions
- 16-bit Stack Pointers and general pointer registers
- Capability to support 32 vectored interrupts - 31 maskable and 1 NMI
- Supports 16 hardware and 16 software traps
- Power Down and Idle power reduction modes
- Hardware support for multi-tasking software

2 Architectural Overview

2.1 Introduction

The Philips XA (eXtended Architecture) has a general purpose register-register architecture to provide the best cost-to-performance trade-off available for a high speed microcontroller using today's technology. Intended as both an upward compatibility path for 80C51 users who need greater performance or more memory, and as a powerful, general-purpose 16-bit controller, the XA also incorporates support for multi-tasking operating systems and high-level languages such as C, while retaining the comprehensive bit-oriented operations that are the hallmark of the 80C51.

This overview introduces the concepts and terminology of the XA architecture in preparation for the detailed descriptions in the following sections of this manual.

2.2 Memory Organization

The XA architecture has several distinct memory spaces. The architecture and the instruction encoding are optimized for register based operations; in addition, arithmetic and logical operations may be done directly on data memory as well. Thus, the XA architecture avoids the bottleneck of having a single accumulator register.

2.2.1 Register File

The register file (Figure 2.1) allows access to 8 words of data at any one time; the eight words are also addressable as 16 bytes. The bottom 4 word registers are "banked". That is, there are four groups of registers, any one of which may occupy the bottom 4 words of the register file at any one time. This feature may be used to minimize the time required for context switching during interrupt service, and to provide more register space for complicated algorithms.

For some instructions –32-bit shifts, multiplies, and divides– adjacent pairs of word registers are referenced as double words.

The upper four words of the register file are not banked. The topmost word register is the stack pointer, while any other word register may be used as a general purpose pointer to data memory.

The entire register file is bit addressable. That is, any bit in the register file (except the 3 unselected banks of the bottom 4 words) may be operated on by bit manipulation instructions.

The XA instruction encoding allows for future expansion of the register file by the addition of 8 word registers. If implemented, these additional registers will be word data registers only and cannot be used as pointers or addressed as bytes.

The overall XA register file structure provides a superset of the 80C51 register structure. For details, refer to the section on 80C51 compatibility.

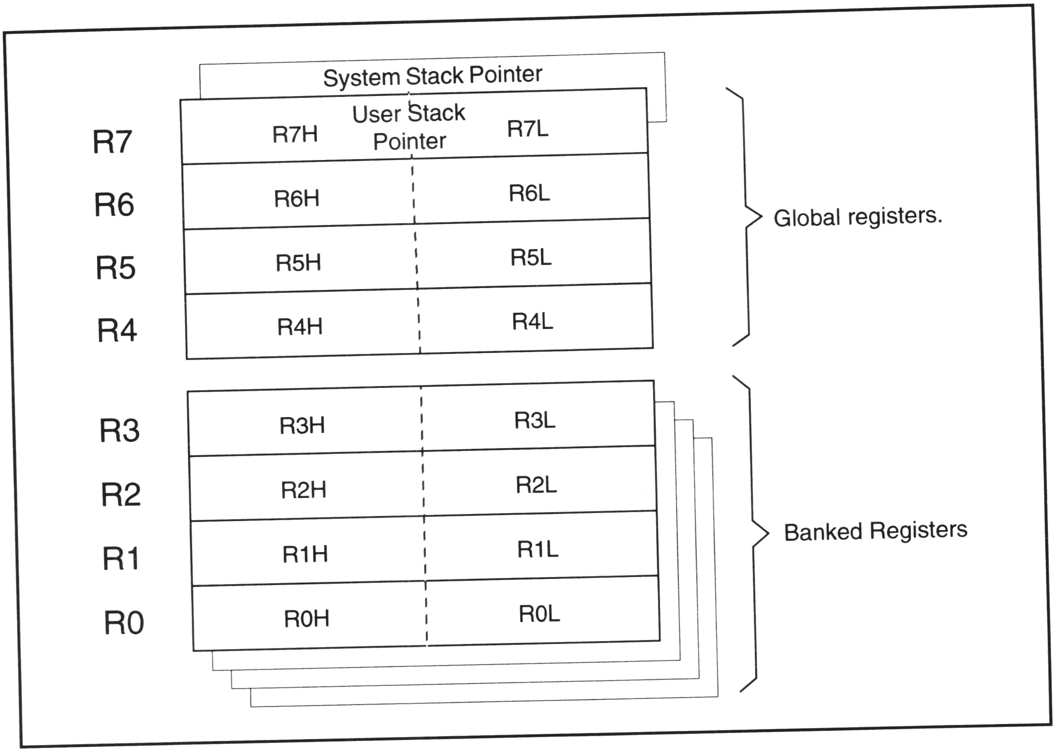

System Stack Pointer
User Stack Pointer
R7
R7H
R7L
R6
R6H
R6L
R5
R5H
R5L
R4
R4H
R4L
Global registers.
R3
R3H
R3L
R2
R2H
R2L
R1
R1H
R1L
R0
R0H
R0L
Banked Registers

Figure 2.1 XA register file diagram

2.2.2 Data Memory

The XA architecture supports a 16 megabyte data memory space with a full 24-bit address. Some derivative parts may implement fewer address lines for a smaller range. The data space beginning at address 0 is normally on-chip and extends to the limit of the RAM size of a particular XA derivative. For addresses above that on a derivative, the XA will automatically roll over to external data memory.

Data memory in the XA is divided into 64K byte segments (Figure 2.2) to provide an intrinsic protection mechanism for multi-tasking systems and to improve performance. Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require large data memories (Figure 2.3).

The XA provides 2 segment registers used to access data memory, the Data Segment register (DS) and the Extra Segment register (ES). Each pointer register is associated with one of the segment registers via the Segment Select (SSEL) register. Pointer registers retain this association until it is changed under program control.

The XA provides flexible data addressing modes. Most arithmetic, logic, and data movement instructions support the following modes of addressing data memory:

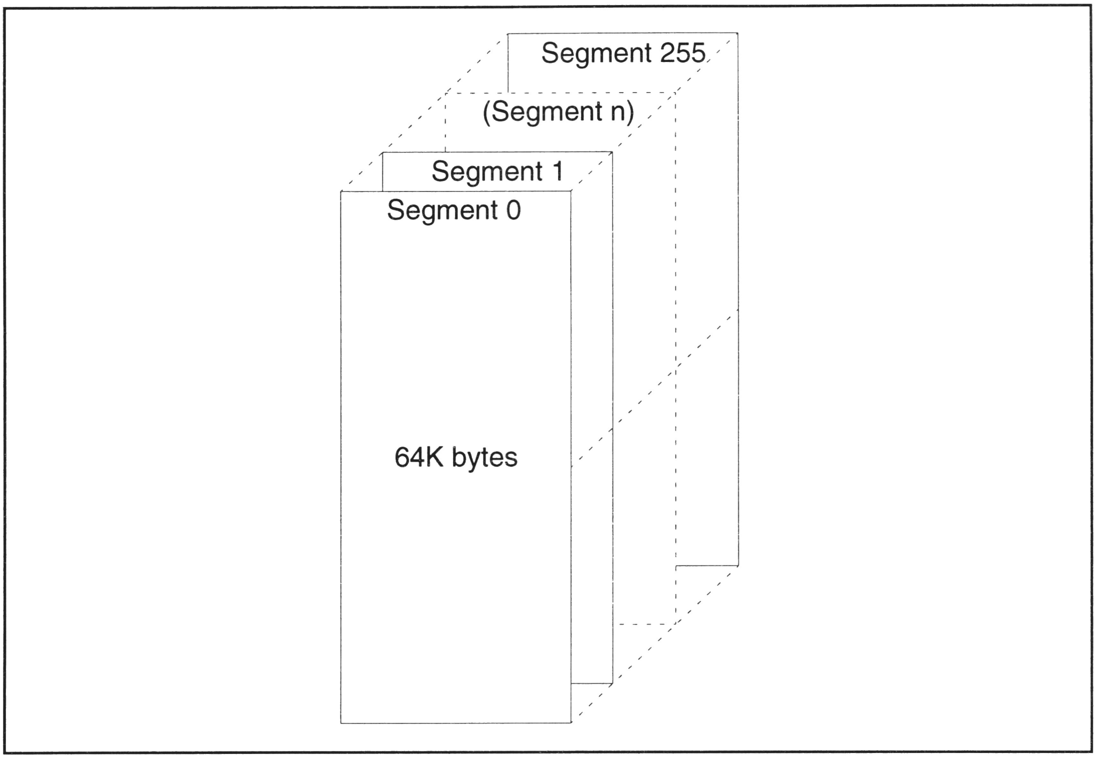

Segment 255
(Segment n)
Segment 1
Segment 0
64K bytes

Figure 2.2 XA data memory segments

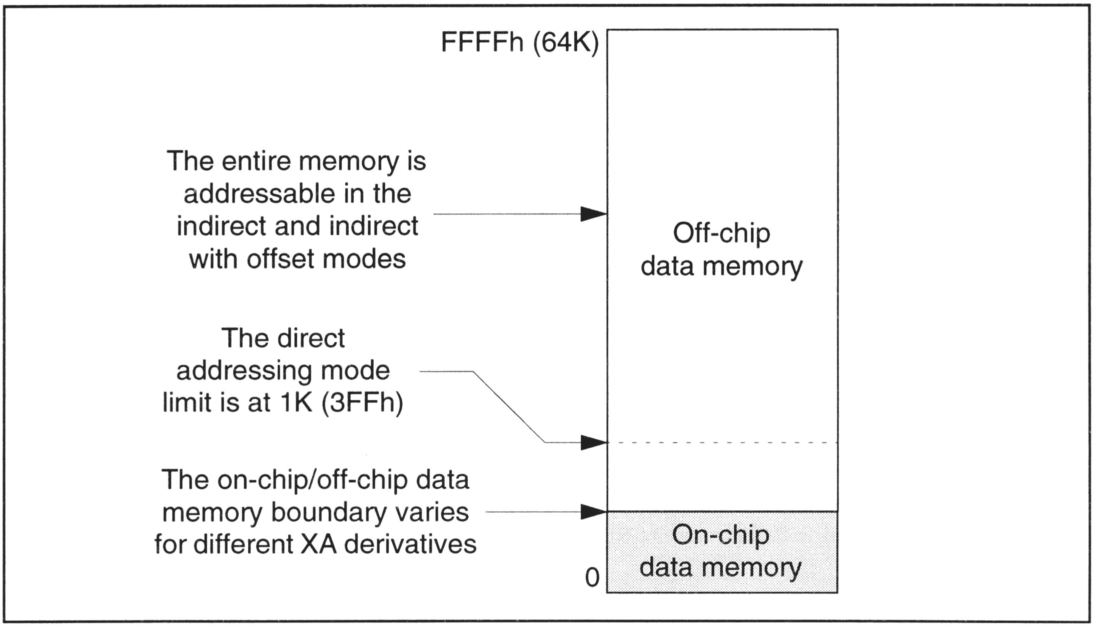

FFFFh (64K)
The entire memory is addressable in the indirect and indirect with offset modes
Off-chip data memory
The direct addressing mode limit is at 1K (3FFh)
The on-chip/off-chip data memory boundary varies for different XA derivatives
On-chip data memory
0

Figure 2.3 Simplified XA data memory diagram

Direct. The first 1K bytes of data on each segment may be accessed by an address contained within the instruction.

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with 16-bits from a pointer register.

Indirect with offset. An 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode allows access into a data structure when a pointer register contains the starting address of the structure. It also allows subroutines to access parameters passed on the stack.

Indirect with auto-increment. The address is formed in the same manner as plain indirect, but the pointer register contents are automatically incremented following the operation.

Data movement instructions and some special purpose instructions also have additional data addressing modes.

The XA data memory addressing scheme provides for upward compatibility with the 80C51. For details, refer to Chapter 9.

2.2.3 Code Memory

The XA is a Harvard architecture device, meaning that the code and data spaces are separate. The XA provides a continuous, unsegmented linear code space that may be as large as 16 megabytes (Figure 2.4). In XA derivatives with on-chip ROM or EPROM code memory, the on-

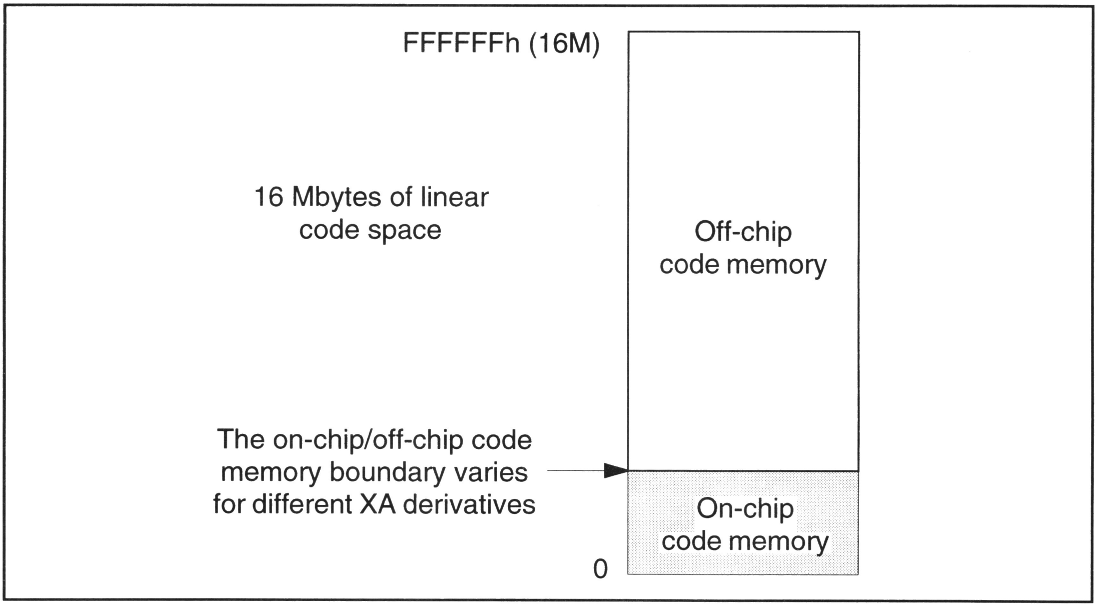

FFFFFFh (16M)
16 Mbytes of linear code space
Off-chip code memory
The on-chip/off-chip code memory boundary varies for different XA derivatives
On-chip code memory
0

Figure 2.4 XA code memory map

chip space always begins at code address 0 and extends to the limit of the on-chip code memory. Above that, code will be fetched from off-chip. Most XA derivatives will support an external bus for off-chip data and code memory, and may also be used in a ROM-less mode, with no code memory used on-chip.

In some cases, code memory may be addressed as data. Special instructions provide access to the entire code space via pointers. Either a special segment register (CS or Code Segment) or the upper 8-bits of the Program Counter (PC) may be used to identify the portion of code memory referenced by the pointer.

2.2.4 Special Function Registers

Special Function Registers (SFRs) provide a means for the XA to access Core registers, internal control registers, peripheral devices, and I/O ports. Any SFR may be accessed by a program at any time without regard to any pointer or segment. An SFR address is always contained entirely within an instruction. See Figure 2.5.

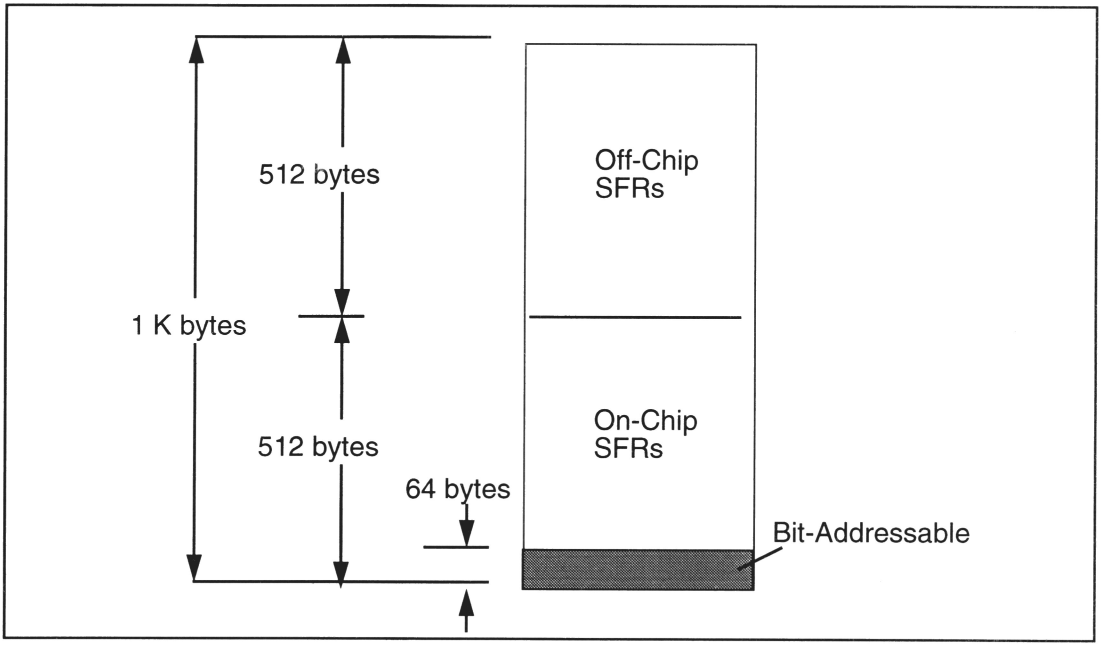


Figure 2.5 SFR Address Space

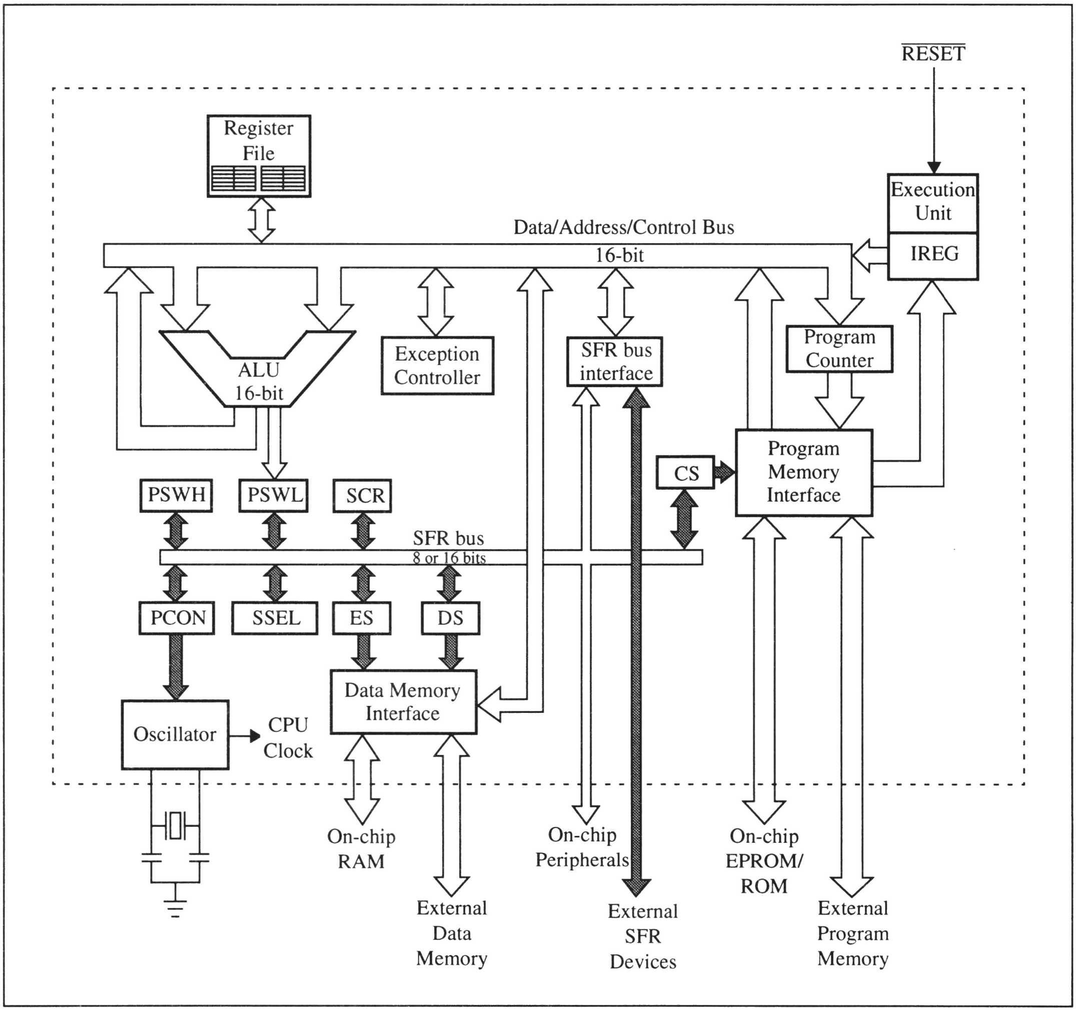
The total SFR space is 1K bytes in size. This is further divided into two 512 byte regions. The lower half is assigned to on-chip SFRs, while the second half is reserved for off-chip SFRs. This allows provides a means to add off-chip I/O devices mapped into the XA as SFRs. Off-chip SFR access is not implemented on all XA derivatives.

On-chip SFRs are implemented as needed to provide control for peripherals or access to CPU features and functions. Each XA derivative may have a different number of SFRs implemented because each has a different set of peripheral functions. Many SFR addresses will be unused on any particular XA derivative.

The first 64 bytes of on-chip SFR space are bit-addressable. Any CPU or peripheral register that allows bit access will be allocated an address within that range.

2.3 CPU

Figure 2.6 shows the XA architecture as a whole. Each of the blocks shown are described in this section.



RESET
Register File
Data/Address/Control Bus
16-bit
Execution Unit
IREG
ALU
16-bit
Exception Controller
SFR bus interface
Program Counter
Program Memory Interface
CS
PSWH
PSWL
SCR
SFR bus
8 or 16 bits
PCON
SSEL
ES
DS
Data Memory Interface
Oscillator
CPU Clock
On-chip RAM
External Data Memory
On-chip Peripherals
External SFR Devices
On-chip EPROM/ ROM
External Program Memory

Figure 2.6 The XA Architecture

2.3.1 CPU Blocks

The XA processor is composed of several functional blocks: Instruction fetch and decode; Execution unit; ALU; Exception controller; Interrupt controller; Register File and core registers; Program memory (ROM or EPROM), Data memory (RAM); SFR and external bus interface; Oscillator; and on-chip peripherals and I/O ports.

Certain functional blocks that exist on most XA derivatives are not part of the CPU core and may vary in each derivative. These are: the external bus interface, the Special Function Register bus (SFR bus) interface, specific peripherals, I/O ports, code and data memories, and the interrupt controller.

CPU Performance Features

The XA core is partially pipelined and performs some CPU functions in parallel. For instance, instruction fetch and decode, and in some cases data write-back, are done in parallel with instruction execution. This partial pipelining gives very fast instruction execution at a very low cost. For instance, the instruction execution time for most register-to-register operations on the XA is 3 CPU clocks, or 100 nanoseconds with a 30 MHz oscillator.

ALU

Data operations in the XA core are accomplished with a 16-bit ALU, providing both 8-bit and 16-bit functions. Special circuitry has been included to allow some 32-bit functions, such as shifts, multiply, and divide.

Core Registers

The XA core includes several key Special Function Registers which are accessed by programs.

The System Configuration Register (SCR) sets up the basic operating modes of the XA. The Program Status Word (PSW) contains status flags that show the result of ALU operations, the register select bits for the four register file banks, the interrupt mask bit, and other system flags. The Data Segment (DS), Extra Segment (ES), and Code Segment (CS) registers contain the segment numbers of active data memory segments. The Segment Select register (SSEL), contains bits that determine which segment register is used by each pointer register in the register file. Bits in the Power Control register (PCON) control the reduced power modes of the processor.

Execution and Control

The Execution and Control block fetches instructions from the code memory and decodes the instructions prior to execution. The XA normally attempts to fetch instructions from the code memory ahead of what is immediately needed by the execution unit. These pre-fetched instructions are stored in a 7 byte queue contained in the fetch and decode unit.

If the fetch unit has instructions in the queue, the execution unit will not have to wait for a fetch to occur when it is ready to begin execution of a new instruction. If a program branch is taken, the queue is flushed and instructions are fetched from the new location. This block also decides whether to attempt instruction fetches from on or off-chip code memory.

The instruction at the head of the queue is decoded into separate functional fields that tell the other CPU blocks what to do when the instruction is executed. These fields are stored in staging registers that hold the information until the next instruction begins executing.

Execution Unit

The execution unit controls many of the other CPU blocks during instruction execution. It routes addressing information, sends read and write commands to the register file and memory control blocks, tells the fetch and decode unit when to branch, controls the stack, and ensures that all of these operations are performed in the proper sequence. The execution unit obtains control information for each instruction from a microcode ROM.

Interrupt Controller

The interrupt controller can receive an interrupt request from any of the sources on a particular XA derivative. It prioritizes these based on user programmable registers containing a priority for each interrupt source. It then compares the priority of the highest pending interrupt (if any) to the interrupt mask bits from the PSW. If the interrupt has a higher priority than the currently running code, the interrupt controller issues a request to the execution unit.

The interrupt controller also contains extra registers for processing software interrupts. These are used to run non-critical portions of interrupt service routines at a decreased priority without risking “priority inversion.”

While the interrupt controller is not part of the XA core, it is present in some form on all XA derivatives.

Exception Controller

The exception controller is similar to the interrupt controller except that it processes CPU exceptions rather than hardware and software interrupt requests. Sources of exceptions are: stack overflow; divide by zero; user execution of an RETI instruction; hardware breakpoint; trace mode; and non-maskable interrupt (NMI).

Exceptions are serviced according to a fixed priority ranking. Generally, exceptions must be serviced immediately since each represents some important event or problem that must be dealt with before normal operation can resume.

The Exception Controller is part of the XA core and is always present.

Interrupt and Exception Processing

Interrupt and exception processing both make use of a vector table that resides in the low addresses of the code memory. Each interrupt and exception has an entry in the vector table that includes the starting address of the service routine and a new PSW value to be used at the beginning of the service routine. The starting address of a service routine must be within the first 64K of code memory.

When the XA services an exception or interrupt, it first saves the return address on the stack, followed by the PSW contents. Next, the PC and the PSW are loaded with the starting address of the appropriate service routine and the new PSW contents, respectively, from the vector table.

When the service routine completes, it returns to the interrupted code by executing the RETI (return from interrupt) instruction. This instruction loads first the PSW and then the Program Counter from the stack, resuming operation at the point of interruption. If more than the PC and PSW are used by the service routine, it is up to that routine to save and restore those registers or other portions of the machine state, normally by using the stack, and often by switching register banks.

Reset

Power up reset and any other external reset of the XA is accomplished via an active low reset pin. A simple resistor and capacitor reset circuit is typically used to provide the power-on reset pulse. the reset pin is a Schmitt trigger input, in order to prevent noise on the reset pin from causing spurious or incomplete resets.

The XA may be reset under program control by executing the RESET instruction. This instruction has the effect of resetting the processor as if an external reset occurred, except that some hardware features that are latched following a hardware reset (such as the state of the EA pin and bus width programming) are not re-latched by a software reset. This distinction is necessary because external circuitry driving those inputs cannot determine that a reset is in progress.

Some XA derivatives also have a hardware watchdog timer peripheral that will trigger an equivalent chip reset if it is allowed to time out.

Oscillator and Power Saving Modes

XA derivatives have an on-chip oscillator that may be used with crystals or ceramic resonators to provide a clock source for the processor.

The XA supports two power saving modes of operation: Idle mode and Power Down mode. Either mode is activated by setting a bit in the Power Control (PCON) register. The Idle mode shuts down all processor functions, but leaves most of the on-chip peripherals and the external interrupts functioning. The oscillator continues to run. An interrupt from any operating source will cause the XA to resume operation where it left off.

The Power Down mode goes one step further and shuts down everything, including the on-chip oscillator. This reduces power consumption to a tiny amount of CMOS leakage plus whatever loads are placed on chip pins. Resuming operation from the power down mode requires the oscillator to be restarted, which takes about 10 milliseconds. Power down mode can be terminated either by resetting the XA or by asserting one of the external interrupts, if one was left enabled when power down mode was entered. In Power Down mode, data in on-board RAM is retained. Further power savings may be made by reducing Vdd in Power Down mode; see the device data sheet for details.

Stack

The processor stack provides a means to store interrupt and subroutine return addresses, as well as temporary data. The XA includes 2 stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP), which correspond to 2 different stacks: the system stack and the user stack. See Figure 2.7. The system stack always resides in the first data memory segment,

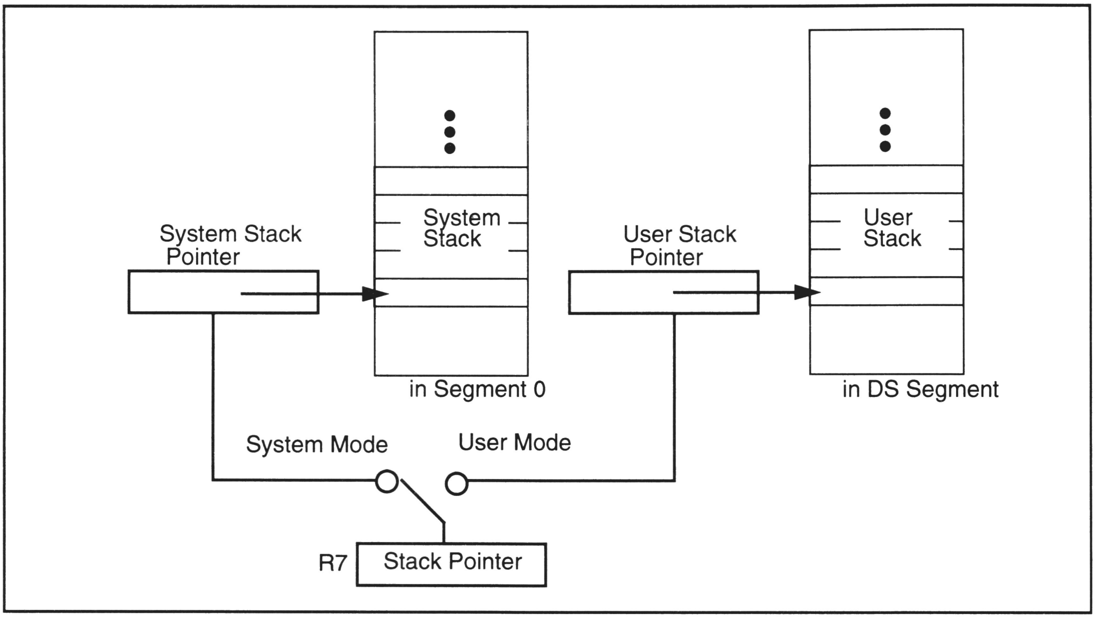


Figure 2.7 XA Stacks

segment 0. The user stack resides in the data memory segment identified by the current value of the data segment (DS) register. Executing code has access to only one of these stacks at a time, via R7. Since each stack resides in a single data memory segment, its maximum size is 64K bytes. The purpose of having two stack pointers will be discussed in more detail in the section on Task Management below.

The XA stack grows downwards, from higher addresses to lower addresses within data memory. The current stack pointer always points to the last item pushed on the stack, unless the stack is empty. Prior to a push operation, the stack pointer is decremented by 2, then data is written to memory. When the stack is popped, the reverse procedure is used. First, data is read from memory, then the stack pointer is incremented by 2. Data on the stack always occupies an even number of bytes and is word aligned in data memory.

Debugging Features

The XA incorporates some special features designed to aid in program and system debugging. There is a software breakpoint instruction that may be inserted in a user's program by a debugger program, causing the user program to break at that point and go to the breakpoint service routine, which can transmit the CPU state so that it can be viewed by the user.

The trace mode is similar to a breakpoint, but is forced by hardware in the XA after the execution of every instruction. The trace service routine can then keep track of every instruction executed by a user program and transmit information about the CPU state to a serial port or other peripheral for display or storage. Trace mode is controlled by a bit in the PSW. The XA is able to alter the trace mode bit whenever an interrupt or exception vector is taken. This gives very flexible use of trace mode, for instance by allowing all interrupts to run at full speed to comply with system hardware requirements, while single stepping through mainline code.

With these two features, a simple monitor debugger routine can allow a user to single step through a program, or to run a program at full speed, stopping only when execution reaches a breakpoint, in either case viewing the CPU state before continuing.

2.4 Task Management

Several features of the XA have been included to facilitate multi-tasking. Multi-tasking can be thought of as running several programs at once on the same processor, with a supervisory program determining when each program, or task, runs, and for how long. Since each task shares the same CPU, the system resources required by each must be kept separate and the CPU state restored when switching execution from one task to another. The problem is much simpler for a microcontroller than it is for a microprocessor, because the code executed by a microcontroller always comes from the same source: the designers of the system it runs on. Thus, this code can be considered to be basically trustworthy and extreme measures to prevent misbehavior are not necessary. The emphasis in the XA design is to protect against simple accidents.

The first step in supporting multi-tasking is to provide two execution contexts, one for the basic tasks –on the XA termed “user mode”– and one for the supervisory program –"system mode.". A program running in system mode has access to all of the processor’s resources and can set up and launch tasks.

Code running in system and user mode use different stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP) respectively. The system stack is always located in the first 64K data memory segment, where it can take advantage of the fast on-chip RAM. The user stack is located within each task’s local data segment, identified by the DS register. The fact that user mode code uses a different stack than system mode code prevents tasks from accidentally destroying data on the system stack and in other task spaces.

Additional protection mechanisms are provided in the form of control bits and registers that are only writable by system mode code. For instance the DS register, that identifies the local data segment for user mode code, is only writable in the system mode. While tasks can still write to the other segment register, the ES register, they cannot write to memory via the ES register unless specifically allowed to do so by the system. The data memory segmentation scheme thus prevents tasks from accessing data memory in unpredictable ways.

Other protected features include enabling of the Trace Mode and alteration of the Interrupt Mask.

The 4 register banks are a feature that can be useful in small multi-tasking systems by using each bank for a different task, including one for system code. This means less CPU state that must be saved during task switching.

2.5 Instruction Set

The XA instruction set is designed to support common control applications. The instruction encoding is optimized for the most commonly used instructions: register to register or register with indirect arithmetic and logic operations; and short conditional and unconditional branches. These instructions are all encoded as 2 bytes. The bulk of XA instructions are encoded as either 2 or 3 bytes, although there are a few 1 byte instructions as well as 4, 5, and 6 byte instructions.

The execution of instructions normally overlaps instruction fetch, and sometimes write-back operations, in order to further speed processing.

2.5.1 Instruction Syntax

The instruction syntax chosen for the XA is similar in many ways to that of the 80C51. A typical XA instruction has a basic mnemonic, such as "ADD", followed by the operands that the operation is to be performed on. The basic syntax is illustrated in Figure 2.8. The direction of operation flow is determined by the order in which operands occur in the source line. For instance, the instruction: "ADD R1, R2" would cause the contents of R1 and R2 to be added together and the result stored in R1. Since R1 and R2 are word registers in the XA, this is a 16-bit operation.

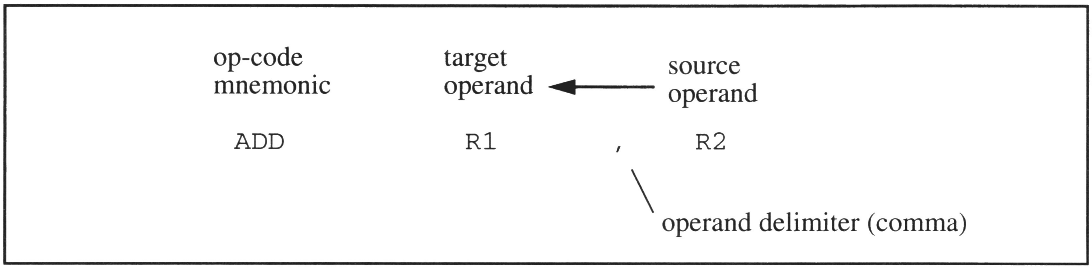


Figure 2.8 Basic Instruction Syntax

An indirect reference (a reference to data memory using the contents of a register as an address) is specified by enclosing the operand in square brackets, as in: "ADD R1, [R2]". See Figure 2.9. This instruction causes the contents of R1 and the data memory location pointed to by R2 (appended to its associated segment register) to be added together and the result stored in R1. Reversing the operand order ("ADD [R2], R1") causes the result to be stored in data memory, as shown in Figure 2.10.

Most instructions support an additional feature called auto-increment that causes the register used to supply the indirect memory address to be automatically incremented after the memory access takes place. The source line for such an operation is written as follows: "ADD R1, [R2+]". As illustrated in Figure 2.11, the auto-increment amount always matches the data size used in the instruction. In the previous example, R2 will have 2 added to it because this was a word operation.

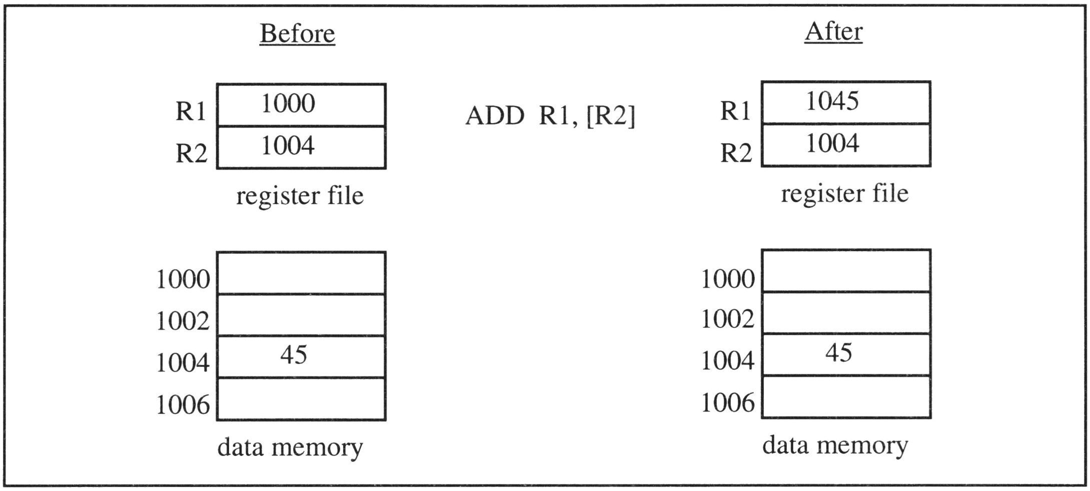

Before
After
R1
1000
R2
1004
ADD R1, [R2]
R1
1045
R2
1004
register file
register file
1000
1002
1004
45
1006
data memory
1000
1002
1004
45
1006
data memory

Figure 2.9 Basic Indirect Addressing Syntax, to register

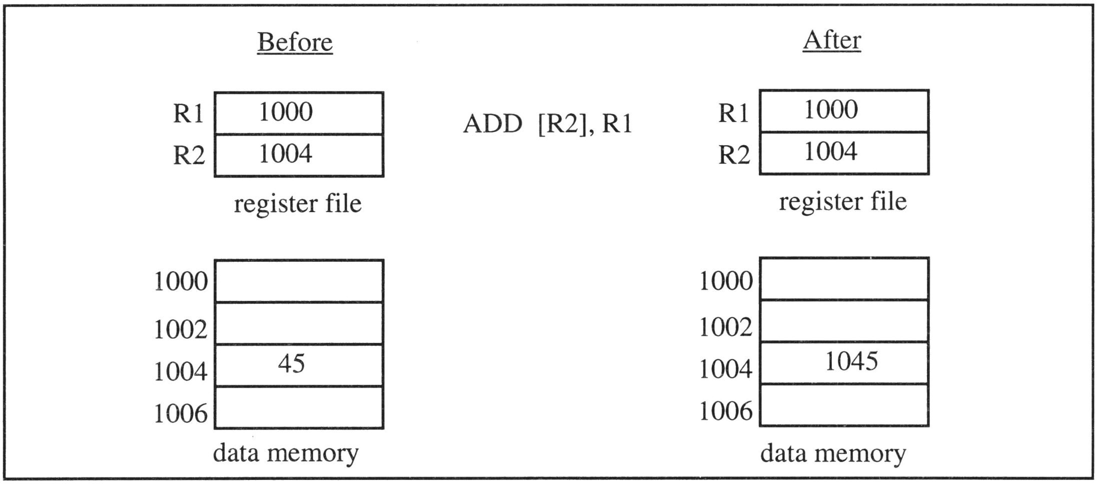

Before
After
R1
1000
R2
1004
ADD [R2], R1
R1
1000
R2
1004
register file
register file
1000
1002
1004
45
1006
data memory
1000
1002
1004
1045
1006
data memory

Figure 2.10 Basic Indirect Addressing Syntax, from Register

Another version of indirect addressing is called indirect with offset mode. In this version, an immediate value from the instruction word is added to the contents of the indirect register in order to form the actual address. This result of the add is 16 bits in size, which is then appended to the segment register for that pointer register. If the offset calculation overflows 16 bits, the overflow is ignored, so the indirect reference always remains on the same segment. The immediate data from the instruction is a signed 8-bit or 16-bit offset. Thus, the range is +127 bytes to -128 bytes for an 8-bit offset, and +32,767 to -32,768 bytes for a 16-bit offset. Note that since the address calculation is limited to 16-bits, the 16-bit offset mode allows access to an entire data segment.

When an instruction requires an immediate data value (a value stored within the instruction itself), it is written using the "#" symbol. For example: "ADD R1, #12" says to add the value 12 to register R1.

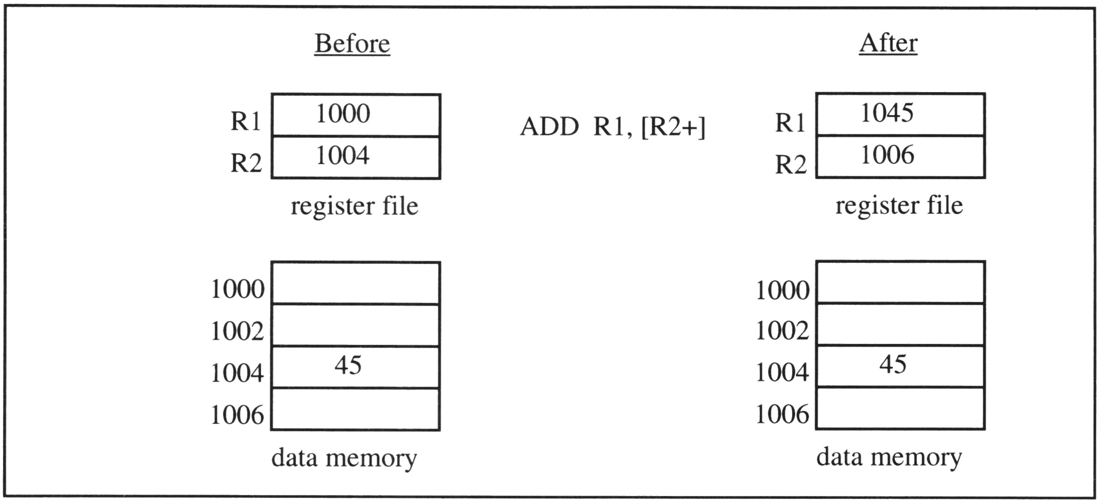


Figure 2.11 Indirect Addressing with Auto-Increment

Since indirect memory references and immediate data values do not implicitly identify the size of the operation to be performed, a few XA instructions must have an operation size explicitly called out. An example would be the instruction: "MOV [R1], #1". The immediate data value does not specify the operation size, and the value stored in memory at the location pointed to by R1 could be either a byte or a word. To clarify the intent of such an instruction, a size identifier is added to the mnemonic as follows: "MOV.b [R1], #1". This tells us that the operation should be performed on a byte. If the line read "MOV.w [R1], #1", it would be a word operation.

If a direct data address is used in an instruction, the address is simply written into the instruction: "ADD 123, R1", meaning to add the contents of register R1 to the data memory value stored at direct address 123. In an actual program, the direct data address could be given a name to make the program more readable, such as "ADD Count, R1".

Operations using Special Function Registers (SFRs) are written in a way similar to direct addresses, except that they are normally called out by their names only: "MOV PSW,#12". Using actual SFR addresses rather than their names in instructions makes the code both harder to read and less transportable between XA derivatives.

Bit addresses within instructions may be specified in one of several ways. A bit may be given a unique name, or it may be referred to by its position within some larger register or entity. An example of a bit name would be one of the status flags in the PSW, for instance the carry ("C") flag. To clear the carry flag, the following instruction could be used: "CLR C". The same bit could be addressed by its position within the PSW as follows: "CLR PSWL.7", where the period (".") character indicates that this is a bit reference. A program may use its own names to identify bits that are defined as part of the application program.

Finally, code addresses are written within instructions either by name or by value. Again, a program is more readable and easier to modify if addresses are called out by name. Examples are: "JMP Loop" and "JMP 124".

2.5.2 Instruction Set Summary

The following pages give a summary of the XA instruction set. For full details, consult Chapter 6.

Basic Arithmetic, Logic, and Data Movement Instructions

The most used operations in most programs are likely to be the basic arithmetic and logic instructions, plus the MOV (move data) instruction. The XA supports the following basic operations:

ADD	Simple addition.
ADDC	Add with carry.
SUB	Subtract.
SUBB	Subtract with borrow.
CMP	Compare.
AND	Logical AND.
OR	Logical OR.
XOR	Exclusive-OR.

These instructions support all of the following standard XA data addressing mode combinations::

Operands	Description
R, R	The source and destination operands are both registers.
R, [R]	The source operand is indirect, the destination operand is a register.
[R], R	The source operand is a register, the destination operand is indirect.
R, [R+]	The source operand is indirect with auto-increment, the destination operand is a register.
[R+], R	The source operand is a register, the destination operand is indirect with auto-increment.
R, [R+offset]	The source operand is indirect with an 8 or 16-bit offset, the destination operand is a register.
[R+offset], R	The source operand is a register, the destination operand is indirect with an 8 or 16-bit offset.
direct, R	The source operand is a register, the destination operand is a direct address.
R, direct	The source operand is a direct address, the destination operand is a register.
R, #data	The source operand is an 8 or 16-bit immediate value, the destination operand is a register.
[R], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect.

Operands	Description
[R+], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with auto-increment.
[R+offset], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with an 8 or 16-bit offset.
direct, #data	The source operand is an 8 or 16 bit immediate value, the destination operand is a direct address.

Other instructions on the XA use different operand combinations. All XA instructions are covered in detail in the Instruction Set section. Following is a summary of other instruction types:Additional arithmetic instructions

Additional arithmetic instructions

ADDS	Add short immediate (4-bit signed value).
NEG	Negate (twos complement).
SEXT	Sign extend.
MUL	Multiply.
DIV	Divide.
DA	Decimal adjust.
ASL	Arithmetic shift left.
ASR	Arithmetic shift right.
LEA	Load effective address.

Additional logic instructions

CPL	Complement (ones complement or logical inverse).
LSR	Logical shift right.
NORM	Normalize.
RL	Rotate left.
RLC	Rotate left through carry.
RR	Rotate right.
RRC	Rotate right through carry.

Other data movement instructions

MOVS	Move short immediate (4-bit signed value).
MOVC	Move to or from code memory.
MOVX	Move to or from external data memory.
PUSH	Push data onto the stack.
POP	Pop data from the stack.
XCH	Exchange data in two locations.

Bit manipulation instructions

SETB	Set (write a 1 to) a bit.
CLR	Clear (write a 0 to) a bit.
MOV	Move a bit to or from the carry flag.
ANL	Logical AND a bit (or its inverse) to the carry flag.
ORL	Logical OR a bit (or its inverse) to the carry flag.

Jump, branch, and call instructions

BR	Branch to code address (plus or minus 256 byte range).
JMP	Jump to code address (range depends on specific JMP variation).
CALL	Call subroutine (range depends on specific CALL variation).
RET	Return from subroutine or interrupt.
Bcc	Conditional branches with 15 possible condition variations.
JB, JNB	Jump if a bit set or not set.
CJNE	Compare two operands and jump if they not equal.
DJNZ	Decrement and jump if the result is not zero.
JZ, JNZ	Jump on zero or not zero (included for 80C51 compatibility).

Other instructions

NOP	No operation (used mainly to align branch targets).
BKPT	Breakpoint (used for debugging).
TRAP	Software trap (used to call system services in a multitasking system).
RESET	Reset the entire chip.

2.6 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices, and initiates code read, data read, or data write strobes. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

As described in section 4.4.4, the initial external bus width is hardware settable, and the XA determines its value (8 or 16 bits) during the reset sequence.

2.6.1 External Bus Signals

The standard XA external bus supports 8 or 16-bit data transfers and up to 24 address lines. The precise number of address lines varies by derivative. The standard control signals and their functions for the external bus are as follows:

Signal name	Function
ALE	Address Latch Enable. This signal directs an external address latch to store a portion of the address for the next bus operation. This may be a data address or a code address.
$\overline{\text{PSEN}}$	Program Store Enable. Indicates that the XA is reading code information over the bus. Typically connected to the Output Enable pin of external EPROMs.
$\overline{\text{RD}}$	Read. The external data read strobe. Typically connected to the $\overline{\text{RD}}$ pin of external peripheral devices.
$\overline{\text{WRL}}$	Write. The low byte write strobe for external data. Typically connected to the $\overline{\text{WR}}$ pin of external peripheral devices. For an 8-bit data bus, this is the only write strobe. For a 16-bit data bus, this strobe applies only to the lower data byte.
$\overline{\text{WRH}}$	Write High. This is the upper byte write strobe for external data when using a 16-bit data bus.
WAIT	Wait. Allows slowing down any type external bus cycle. When asserted during a bus operation, that operation waits for this signal to be de-asserted before it is completed.

2.6.2 Bus Configuration

The standard XA bus is user configurable in several ways. First, the bus size may be configured to either 8 bits or 16 bits. This may be configured by the logic level on a pin at reset, or under firmware control (if code is initially executed from on-chip code memory) prior to any actual external bus operations. As on the 80C51, the $\overline{\text{EA}}$ pin determines whether or not on-chip code memory is used for initial code fetches.

Second, the number of address lines may be configured in order to make optimal use of I/O ports. Since external bus functions are typically shared with I/O ports and/or peripheral I/O functions, it is advantageous to set the number of address lines to only what is needed for a particular application, freeing I/O pins for other uses.

2.6.3 Bus Timing

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, $\overline{\text{PSEN}}$ width, $\overline{\text{RD}}$ and $\overline{\text{WRL}}/\overline{\text{WRH}}$ width, and data hold time from $\overline{\text{WRL}}/\overline{\text{WRH}}$. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

The following figures show the basic sequence of events and timing of typical XA bus accesses. For more detailed information, consult Section 7 and the device data sheet.

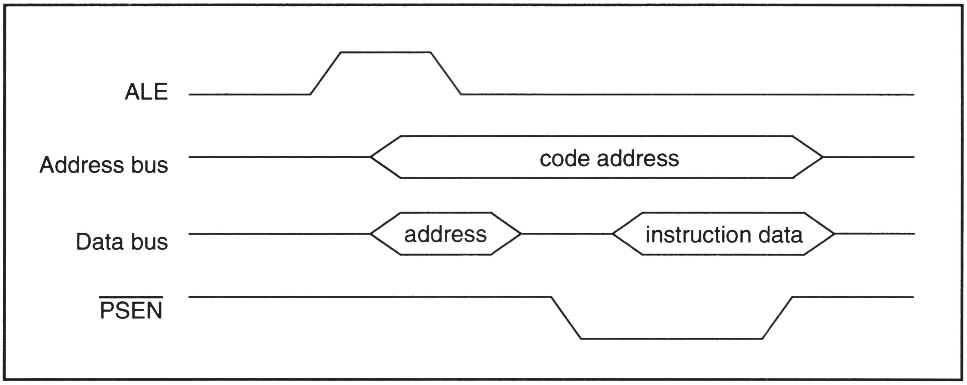

ALE
Address bus
code address
Data bus
address
instruction data
$\overline{\text{PSEN}}$

Figure 2.12 Typical External Code Read.

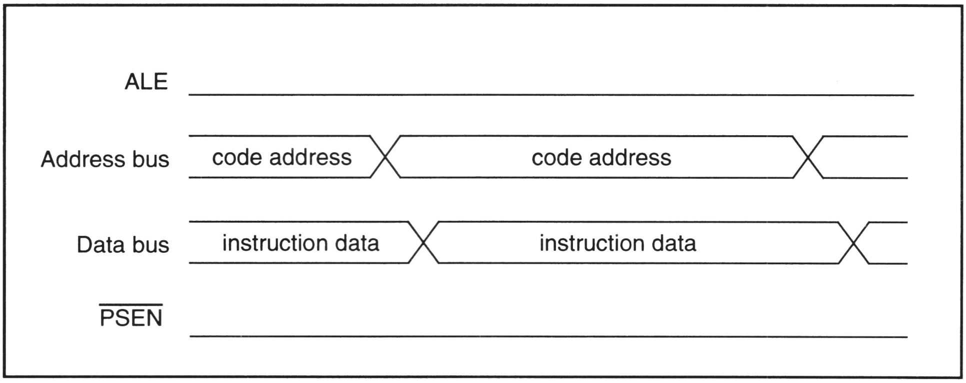

ALE
Address bus
code address
code address
Data bus
instruction data
instruction data
$\overline{\text{PSEN}}$

Figure 2.13 Optimized (Sequential Burst) External Code Read.

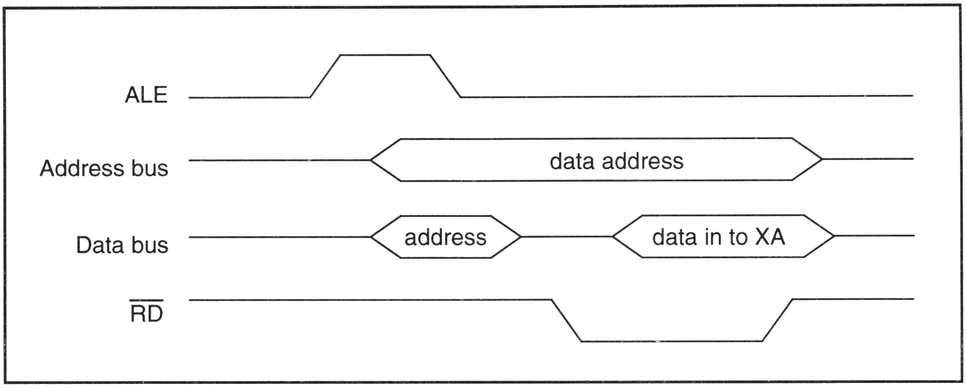

ALE
Address bus
data address
Data bus
address
data in to XA
RD

Figure 2.14 Typical External Data Read.

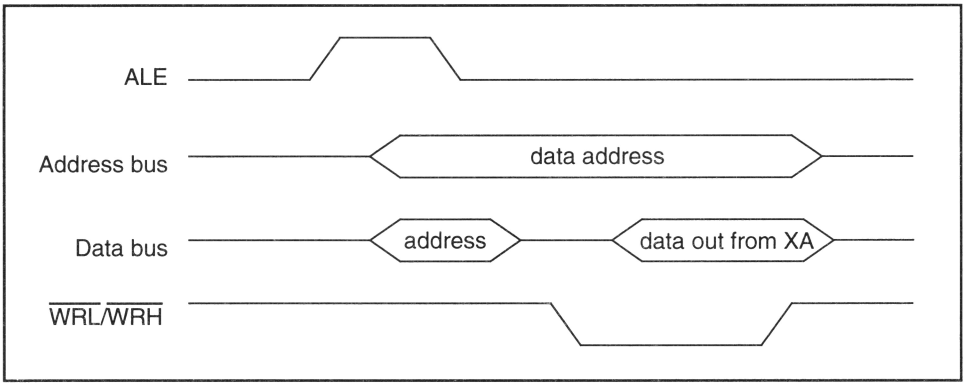

ALE
Address bus
data address
Data bus
address
data out from XA
WRL/WRH

Figure 2.15 Typical External Data Write.

2.7 Ports

Standard I/O ports on the XA have been enhanced to provide better versatility and programmability than was previously available in the 80C51 and most of its derivatives. Access to the I/O ports from a program is through SFR addresses assigned to those ports. Ports may be read and written is the same manner as any other SFR.

The XA provides more flexibility in the use of I/O ports by allowing different output configurations. See Figure 2.16. Port outputs may be programmed to be quasi-bidirectional (80C51 style ports), open drain, push-pull, and high impedance (input only).

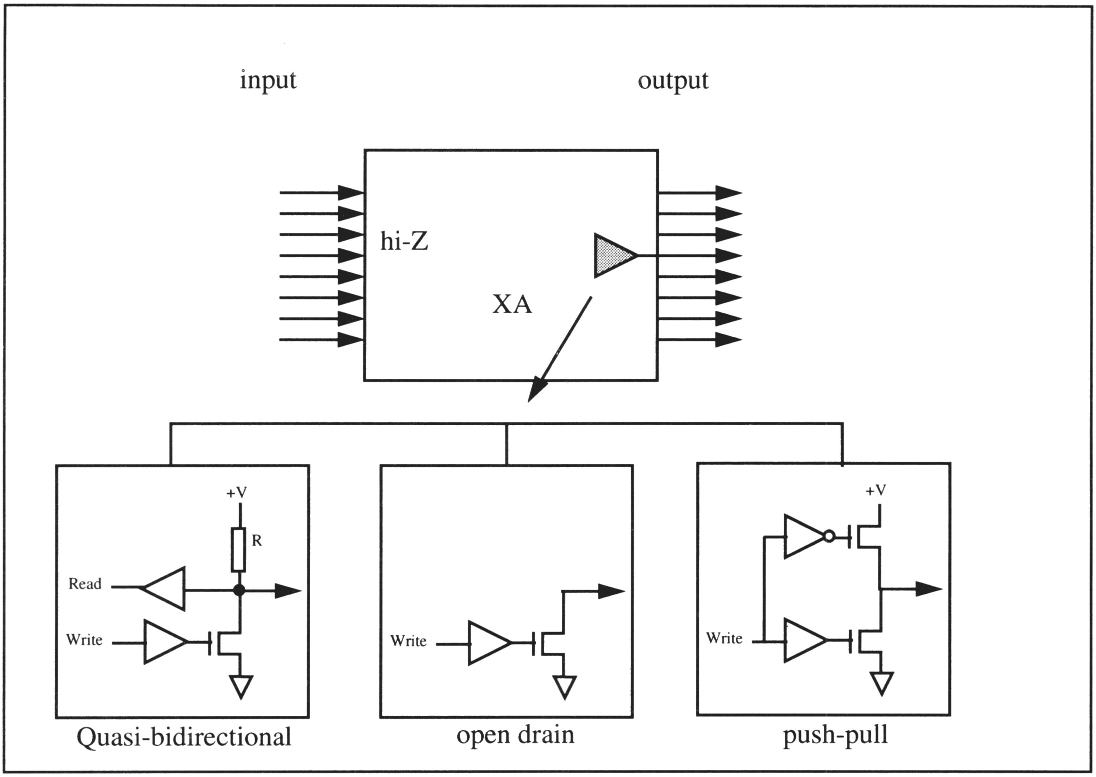


Figure 2.16 XA Port Pins with Driver Option Detail

2.8 Peripherals

The XA CPU core is designed to make derivative design fast and easy. Peripheral devices are not part of the core, but are attached by means of a Special Function Register bus, called the SFR bus, which is distinct from the CPU internal buses. So, a new XA derivative may be made by designing a new SFR bus compatible peripheral function block, if one does not already exist, then attaching it to the XA core.

2.9 80C51 Compatibility

The 80C51 is the most extensively designed-in 8-bit microcontroller architecture in the world, and a vast amount of public and private code exists for this device family. For customers who use the 80C51 or one of its derivatives, preservation of their investment in code development is an important consideration. By permitting simple translation of source code, the XA allows existing 80C51 code to be re-used with this higher-performance 16-bit controller. At the same time, the XA hardware was designed with the clear goal of upward compatibility. 80C51 designs may be migrated to the XA with very few changes necessary to software source or hardware.

The XA provides an 80C51 Compatibility Mode, which essentially replicates the 80C51 register architecture for the best possible upward compatibility. In the alternative Native Mode, the XA operates as an optimized 16-bit microcontroller incorporating the best conceptual features of the original 80C51 architecture.

Many trade-offs and considerations were taken into account in the creation of the XA architecture. The most important goal was to make it possible for a software translator to convert 80C51 assembler source code to XA source code on a 1:1 basis, i.e., one XA instruction for one 80C51 instruction.

Some specific compatibility issues are summarized in the following two sections. See Chapter 9 for a complete description of compatibility.

2.9.1 Software Compatibility

Several basic goals were observed in order to design 80C51 software compatibility for the XA, while avoiding over-complicating the XA design. Following are some key issues for XA software:

- Instruction mapping. Each 80C51 instruction translates into one XA instruction. Multi-instruction combinations that could result in problems if split by an interrupt were avoided as much as possible. Only one 80C51 instruction does not have a one-to-one direct replacement with an XA instruction (this instruction, XCHD, is extremely rarely used).

- "As-is" instructions. Most XA instructions are more powerful supersets of 80C51 instructions. Where this was not possible, the original 80C51 instruction is included "as-is" in the XA instruction set.

- Timing. Instruction timing must necessarily change in order to improve performance. The XA does not attempt to retain timing compatibility with the 80C51; rather, the design simply maximizes instruction execution speed. When 80C51 code that is timing critical is translated to the XA, the user must re-analyze the timing and make adjustments.

- SFR Access. Translation of SFR accesses is usually simple, since SFRs are normally referenced by name. Such references are simply retained in the translated XA code. If program source code from a specific 80C51 derivative references an SFR by its address, the translator can directly substitute the appropriate XA SFR, provided both the 80C51 and the XA derivative are correctly identified to the translator.

2.9.2 Hardware Compatibility

The key goal for hardware was to produce a familiar architecture with a good deal of upward compatibility.

- Memory Map. A major consideration in hardware compatibility of the XA with the 80C51 is the memory map. The XA approaches this issue by having each memory area (registers, data memory, code memory, stack, SFRs) be a superset of the corresponding 80C51 area.

- Stack. One area where a functional change could not be avoided is in the use of the processor stack. Due to the fact that the XA supports 16-bit operations in memory, it was necessary to change the direction of stack growth to downward –the standard for 16-bit processors– in order to match stack usage with efficient access of 16-bit variables in memory. This is an important consideration for support of high-level language compilers such as C.

- Pin-for-pin compatibility. XA derivatives are not intended to be exactly pin-compatible with other 80C51 derivatives that have similar features. Many on-chip XA peripherals, for example, have improved capabilities, and maintaining pin-for-pin compatibility would limit access to these capabilities. In general, peripherals have been made upward compatible with the original 80C51 devices, and most enhancements are added transparently. In these cases, 80C51 code will operate correctly on the 80C51 functional subset.

- Bus Interface. The external bus on the XA is an example of a trade-off between 80C51 compatibility and performance. In order to provide more flexibility and maximum performance, the 80C51 bus had to be changed somewhat. The differences are described in detail in the section on the external bus.

3 XA Memory Organization

3.1 Introduction

The memory space of XA is configured in a Harvard architecture which means that code and data memory (including sfrs) are organized in separate address spaces. The XA architecture supports 16 Megabytes (24-bit address) of both code and data space. The size and type of memory are specific to an XA derivative.

The XA supports different types of both code and data memory e.g.,code memory could be Eprom, EEProm, OTP ROM, Flash, and Masked ROM whereas data memory could be RAM, EEProm or Flash.

This chapter describes the XA Memory Organization of register, code, and data spaces; how each of these spaces are accessed, and how the spaces are related.

3.2 The XA Register File

The XA architecture is optimized for arithmetic, logical, and address-computation operations on the contents of one or more registers in the XA Register File.

3.2.1 Register File Overview

The XA architecture defines a total of 16 word registers in the Register File:
In the baseline XA core, only R0 through R7 are implemented. These registers are available for unrestricted use except R7– which is the XA stack pointer, as illustrated in Figure 3.1. In effect, the XA registers provide users with at least 7 distinct “accumulators” which may be used for all operations. As will be seen below, the XA registers are accessible at the bit, byte, word, and doubleword level.

Additional global registers, R8 through R15, are reserved and may be implemented in specific XA derivatives. These registers, when available, are equivalent to R0 through R7 except byte access and use as pointers will not be possible (only word, double-word, and bit-addressable). The Register File is independent of all other XA memory spaces (except in Compatibility Mode; see chapter 9).

Register File Detail

Figure 3.2 describes R0 through R7 in greater detail.

Byte, Word, and Doubleword Registers
All registers are accessible as bits, bytes, words, and –in a few cases– doublewords. Bit access to registers is described in the next section. As for byte and word accesses, R1 –for example– is a word register that can be word referenced simply as “R1”. The more significant byte is labeled as “R1H” and the less significant byte of R1 is referenced as “R1L”. Double-word registers are always formed by adjacent pairs of registers and are used for 32 bit shifts, multiplies, and divides. The pair is referenced by the name of the lower-numbered register (which contains the

less significant word), and this must have an even number. Thus valid double-register pairs are (R0,R1), (R2,R3), (R4,R5) and (R6, R7).

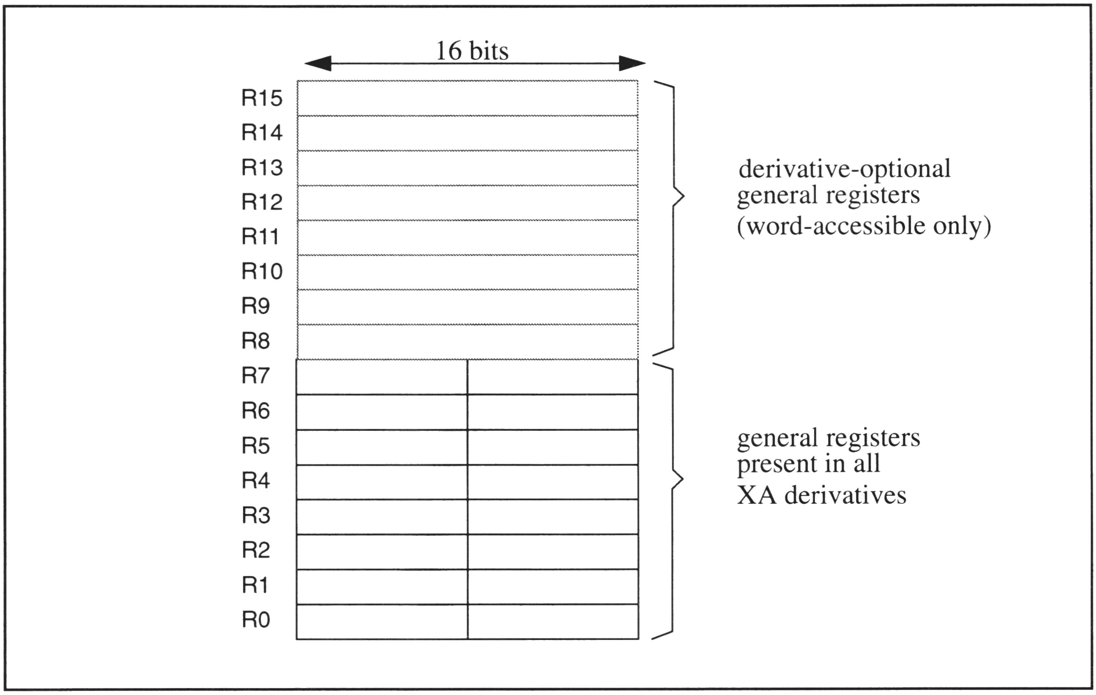

16 bits
R15
R14
R13
R12
R11
R10
R9
R8
R7
R6
R5
R4
R3
R2
R1
R0
derivative-optional
general registers
(word-accessible only)
general registers
present in all
XA derivatives

Figure 3.1 XA Register File Overview

As described in section 4.7, there are two stack pointers, one for user mode and another for system mode. At any given instant only one stack pointer is accessible and its value is in R7. When PSW.SM is 0, user mode is active and the USP is accessible via R7. When PSW.SM is 1, the XA is operating in system mode, and SSP is in R7. (Note however, as described in Chapter 4, all interrupts save stack frames on the system stack, using the SSP, regardless of the current operating mode.)

There are four distinct instances of registers R0 through R3. At any given time, only 1 set of the 4 banks is active, referenced as R0 through R3, and the contents of the other banks are inaccessible. This allows high-speed context-switching, for example, for interrupt service routines. **PSW** bits **RS1** and **RS0** select the active register bank:

RS1	RS0	visible register bank
0	0	bank 0
0	1	bank 1
1	0	bank 2
1	1	bank 3

PSW.RSn are writable when the XA is operating in system or user mode, and programs running in either mode may explicitly change these bits to make selected banks visible one at a time. More commonly, the interrupt mechanism, as described in Chapter 4, provides automatic implicit register bank switching so interrupt handlers may immediately begin operating in a reserved register context.

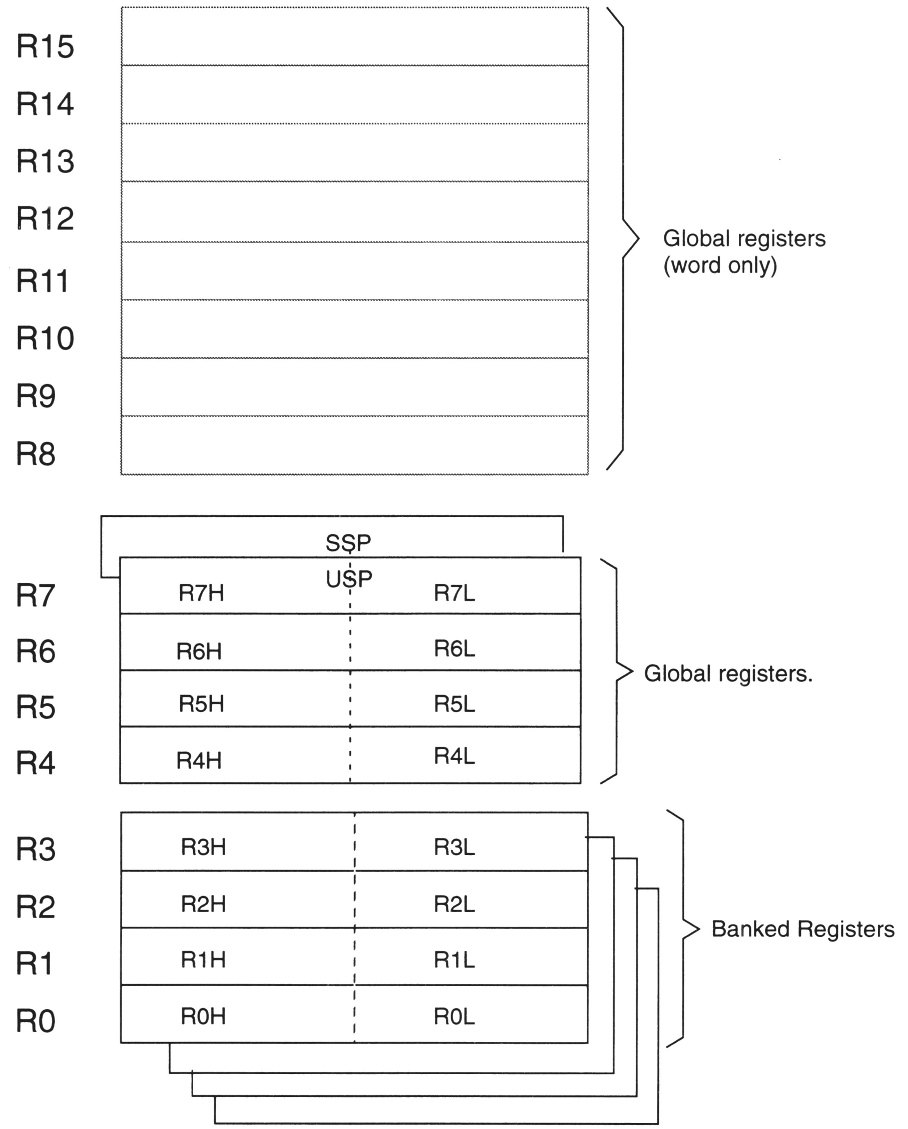

R15
R14
R13
R12
R11
R10
R9
R8
Global registers
(word only)
SSP
USP
R7
R7H
R7L
R6
R6H
R6L
R5
R5H
R5L
R4
R4H
R4L
Global registers.
R3
R3H
R3L
R2
R2H
R2L
R1
R1H
R1L
R0
R0H
R0L
Banked Registers

Figure 3.2 XA Register File

Bit Access to Registers

The XA Registers are all bit addressable. Figure 3.3 shows how bit addresses overlie the basic register file map. In general, absolute bit references as given in this map are unnecessary. XA software development tools provide symbolic access to bits in registers. For example, bit 7 may be designated as "R0.7" with no ambiguity

Bit references to banked registers R0 through R4 access the currently accessible register bank, as set by **PSW** bits **RS1**, **RS0** and the currently selected stack pointer USP or SSP. The unselected registers are inaccessible..

	RnH								RnL							
R15	FF	FE	FD	FC	FB	FA	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0
R14	EF	EE	ED	EC	EB	EA	E9	E8	E7	E6	E5	E4	E3	E2	E1	E0
								⋮								
R7	7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70
R6	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	60
R5	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50
R4	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40
R3	3F	3E	3D	3C	3B	3A	39	38	37	36	35	34	33	32	31	30
R2	2F	2E	2D	2C	2B	2A	29	28	27	26	25	24	23	22	21	20
R1	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10
R0	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00

Figure 3.3 Bit Address to Registers

3.3 The XA Memory Spaces

The XA divides physical memory into program and data memory spaces. Twenty-four address bits, corresponding to a 16MB address space, are defined in the XA architecture. In any given XA implementation, fewer than all twenty-four address bits may actually be used, and there is provision for a small-memory mode which uses only 16-bit addresses; see Chapter 4.

Code and data memory may be on-chip or external, depending on the XA variant and the user implementation. Whether a specific region is on-chip or external does not, in general, affect access to the memory.

3.3.1 Bytes, Words, and Alignment

XA memory is addressed in units of *bytes*, where each byte consists of 8 bits. A *word* consists of two bytes, and the word storage order is “Little-Endian”, that is, the less significant byte of word data is located at a lower memory address. See Figure 3.4.

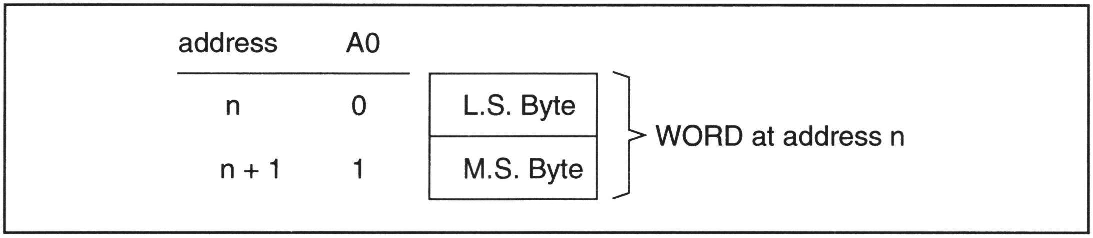


Figure 3.4 Memory byte order

Any word access must be aligned at an even address (Address bit A0=0). If an odd-aligned word access is attempted the word at the next-smallest even address will be accessed, that is, A0 will be set to 0.

The external XA memory spaces may be accessed in byte or word units but the hardware access method does not affect the even alignment restriction on word accesses.

3.4 Data Memory

The data memory space starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. As will be described below, the data memory space is segmented into 256 segments of 64K bytes each. *External Data Memory* starts at the first address following the highest *Internal Data Memory* location. In general, at least 512 bytes of Internal Data Memory, starting at location 0, will be provided in all XA implementations; however, there is no inherent minimum or maximum architectural limitation on Internal Data Memory.

3.4.1 Alignment in Data Memory

There are no data memory alignment restrictions except that placed on word accesses to all memory: Words must be fetched from even addresses. An attempt to fetch a word at an odd address will fetch a word from the preceding even address.

3.4.2 External and Internal Overlap

If External Data Memory is placed by external logic at addresses that overlaps Internal Data Memory, the Internal Data Memory generally takes precedence. The overlapped portion of the External memory may be accessed only by using a form of the MOVX instruction; see Chapter 6. The use of MOVX always forces external data memory fetch in XA. For non-overlapped portion of external data memory, no MOVX is required.

3.4.3 Use and Read/Write Access

Data memory is defined as read-write, and is intended to contain read/write data. It is logically impossible to execute instructions from XA Data Memory. It is possible, and a common practice, to add logic to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such a modified Harvard architecture, implemented with external logic, it is possible –but not recommended– to write self-modifying XA code. No such overlap is possible for internal data memory.

3.4.4 Data Memory Addressing

XA data memory addressing is optimized for the needs of embedded processing. Data memory in the XA is divided into 64K byte segments. This provides an intrinsic protection mechanism for multitasking applications and improves performance by requiring fewer address bits for localized accesses.

Addressing through Segment Registers

Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require full use of the XA 16 Mbyte address space. Two segment registers are defined in the XA architecture for use in accessing data memory, the Data Segment Register (**DS**), and the Extra Segment Register (**ES**). As user stacks are located in the segment specified by **DS**, it is probably most convenient to address user data structures through **ES**. Each pointer register, namely R0 through R6, is associated with one of the segment registers via the Segment Select (**SSEL**) register as illustrated in Figure 3.5.

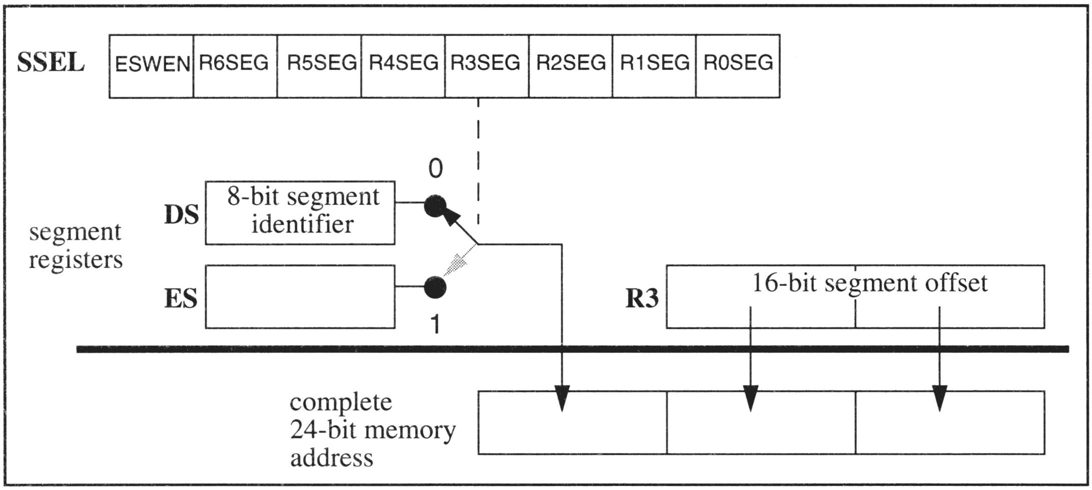


Figure 3.5 Address generation

A 0 in the SSEL bit corresponding to the pointer register selects DS (default on RESET) and 1 selects the ES. For example, when R3 contains a pointer value, the full 24 bit address is formed by concatenating DS or ES, as determined by the state of SSEL bit 3, as the most significant 8 bits. As a consequence of segmented addressing, the XA data memory space may be viewed as 256 segments of 64K bytes each (Figure 3.6).

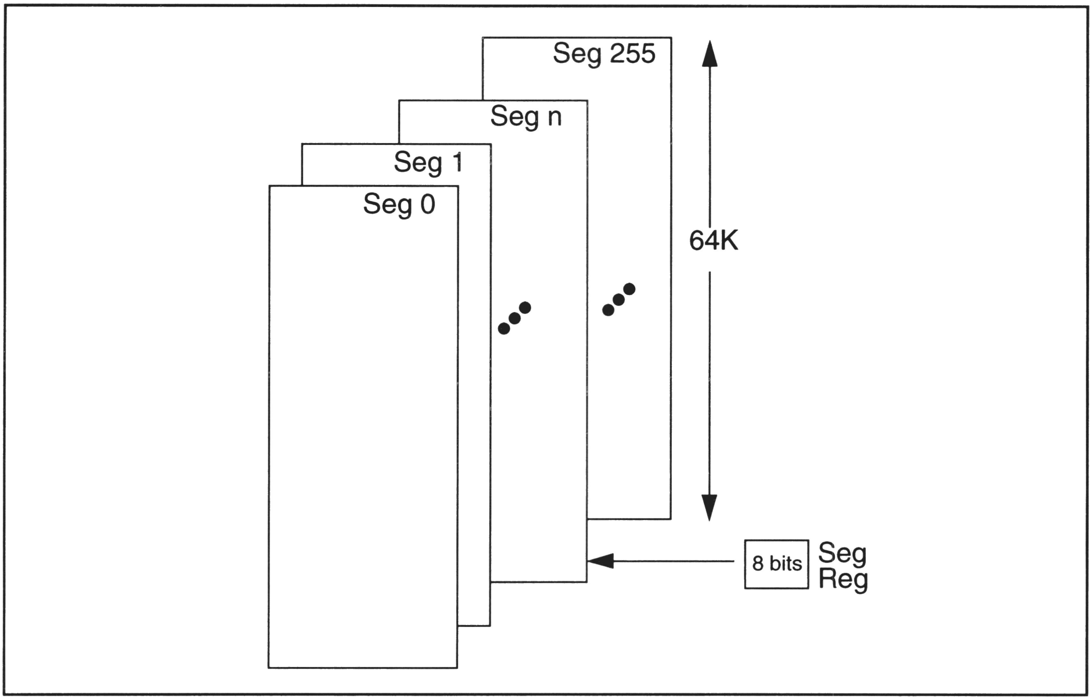


Figure 3.6 Data memory segmentation

Addressing Modes

The XA provides flexible data addressing modes. Arithmetic, logic, and data movement instructions generally support the following data memory access:

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with a 16-bit pointer in a register.

Direct. The first 1K bytes of data in each segment may be accessed by an address contained within the instruction. *Indirect with offset.* A signed byte/word offset contained within the instruction is added to the contents of a pointer register, and the result is concatenated with the 8-bit segment register DS to produce a complete 24-bit address.

Indirect with auto-increment. Indirect addresses are formed as above and the pointer register contents are automatically incremented.

Bi*t-level.* Bit-level addresses are absolute references to specific bits.

Data move instructions and some special purpose instructions also have additional data addressing modes as described in Chapter 6.

Indirect Addressing

The entire 16 MByte address space is accessible via register-indirect addressing with a segment register, as illustrated by Figure 3.7 (Note that for simplicity, this figure omits showing how the Extra Segment or Data Segment Register is chosen using **SSEL**.).

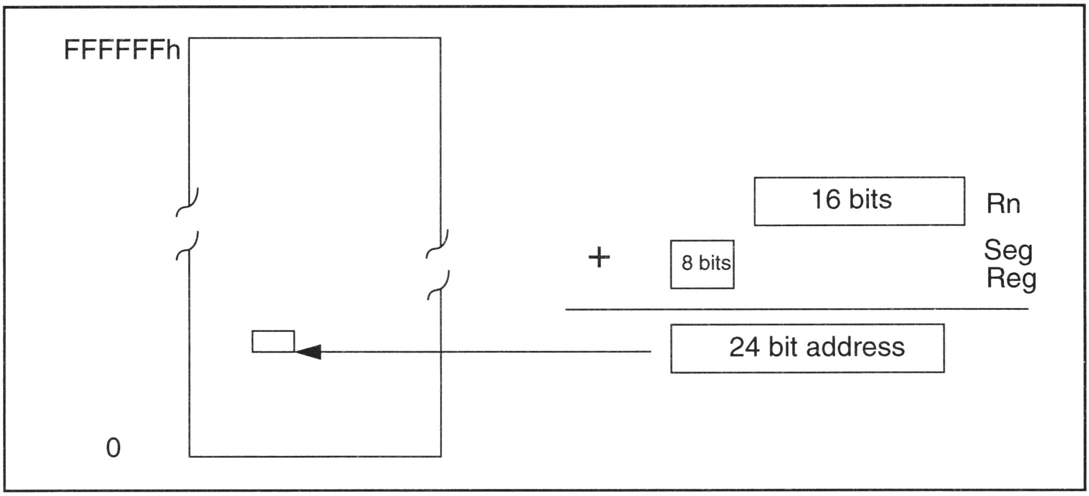


Figure 3.7 Indirect Access to 24 Bit Address Space

Indirect addressing with an offset is a variant of general indirect addressing in which an 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode gives access to data structures when a pointer register contains the starting address of the structure. It also supports stack-based parameter passing.

Indirect addressing with autoincrement is another variant of indirect addressing in which the pointer register contents are automatically incremented following the operation. When the operand is a byte, the increment is one; when the operand is a word, the increment is 2. Using indirect addressing with auto-increment provides a convenient method of traversing data structures smaller than 64K bytes. For data structures exceeding 64K bytes in length, the program code must explicitly adjust the segment register at page boundaries.

Address generation in these two modes of indirect addressing is illustrated inFigures 3.8 and 3.9. When using indirect addressing care is necessary to avoid accessing a word quantity at an odd address. This will result in an access using the next-lower even address, which is generally not desirable. Note that the indirect addressing with an offset will be successful in this case as long as the final, effective address is even. That is, both the base address and the offset may be odd.

Direct Addressing

The first 1K of each segment is directly addressable. Address generation for the direct address mode is summarized in Figure 3.10. Segment register DS is always used.
Direct data-reference instructions encode a maximum of 10 address bits, which are zero extended to sixteen bits and concatenated with DS to form an absolute 24 bit address. In all segments, direct addressing can be used to access any byte in the first 1K bytes of the segment.

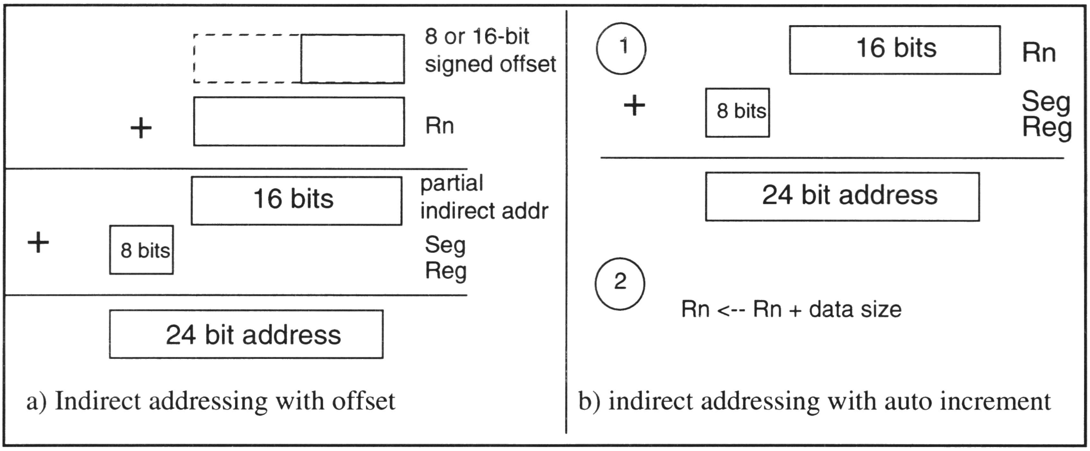

8 or 16-bit
signed offset
+
Rn
16 bits
partial
indirect addr
+
8 bits
Seg
Reg
24 bit address
a) Indirect addressing with offset
1
16 bits
Rn
+
8 bits
Seg
Reg
24 bit address
2
Rn <-- Rn + data size
b) indirect addressing with auto increment

Figure 3.8 Indirect Addressing

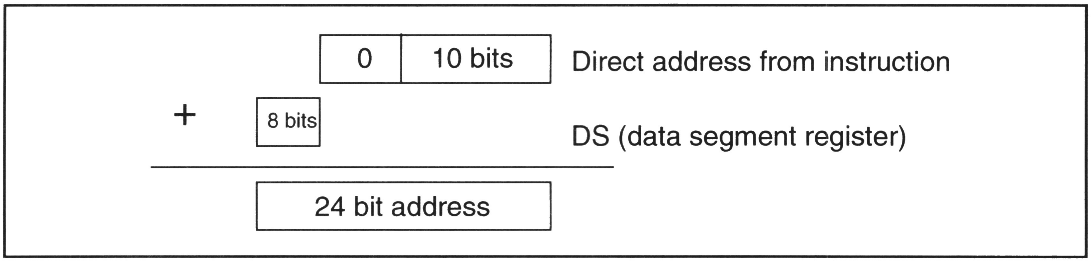

0
10 bits
Direct address from instruction
+
8 bits
DS (data segment register)
24 bit address

Figure 3.9 Direct address generation

SFR Addressing

A 1K portion of the direct address space, addresses 400h through 7FFh, is reserved for SFR addresses. The SFR address space uses a portion of the direct address space, but represents a completely distinct logical area that is not related to the data memory segmentation scheme. See section 3.6 for a complete description of SFR access.

Bit Addressing

Thirty-two bytes of each segment of data memory are also bit-addressable, starting at offset 20h in the segment addressed by the DS register. Address generation for bit addressing in the data memory space is shown in Figure 3.10. As described in chapter 6, bits are encoded in instructions as 10 bits. Figure 3.11 shows the bit addresses as they appear in memory .

3.5 Code Memory

Code memory starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. *External Code Memory* (off-chip) starts at the first address following the highest *Internal Code Memory* (on-chip) location, if any. If code memory is present on-chip, it always starts at location 0.

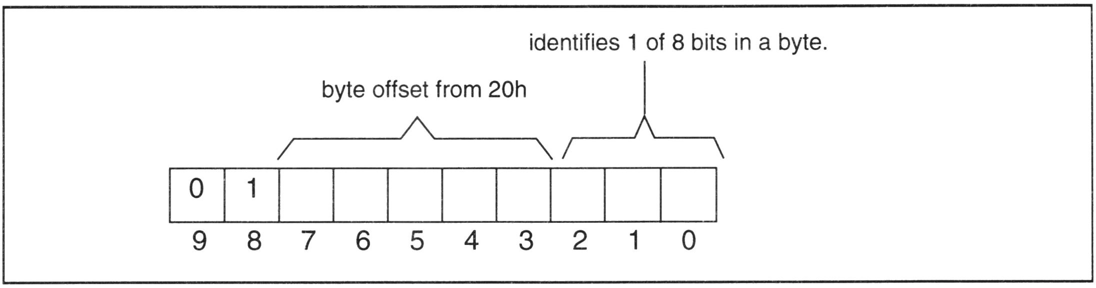
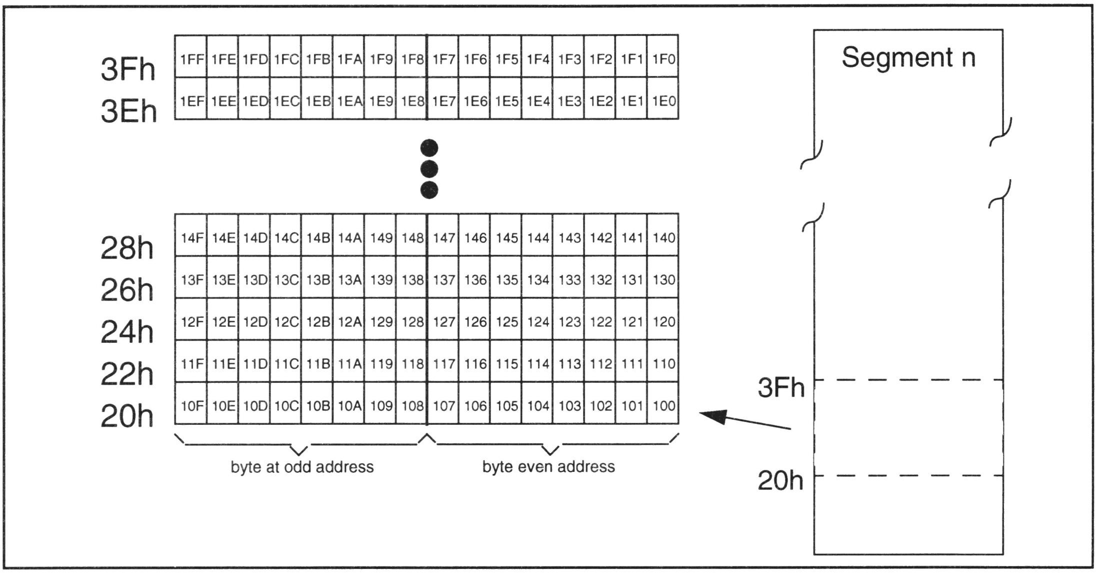


Figure 3.10 Bit address generation in direct memory space



3Fh	1FF	1FE	1FD	1FC	1FB	1FA	1F9	1F8	1F7	1F6	1F5	1F4	1F3	1F2	1F1	1F0
3Eh	1EF	1EE	1ED	1EC	1EB	1EA	1E9	1E8	1E7	1E6	1E5	1E4	1E3	1E2	1E1	1E0
28h	14F	14E	14D	14C	14B	14A	149	148	147	146	145	144	143	142	141	140
26h	13F	13E	13D	13C	13B	13A	139	138	137	136	135	134	133	132	131	130
24h	12F	12E	12D	12C	12B	12A	129	128	127	126	125	124	123	122	121	120
22h	11F	11E	11D	11C	11B	11A	119	118	117	116	115	114	113	112	111	110
20h	10F	10E	10D	10C	10B	10A	109	108	107	106	105	104	103	102	101	100

Figure 3.11 Direct memory bit addressing

3.5.1 Alignment in Code Memory

As instructions are variable in length, from 1 to 6 bytes (see Chapter 6), instructions in code memory can be located at odd addresses. As described in Chapter 6, instruction branch targets, i.e., targets of jumps, calls, branches, traps, and interrupts must be aligned on an even address.

3.5.2 External and Internal Overlap

If External Code Memory is placed by external logic at locations that overlap Internal Code Memory, the Internal Code Memory takes precedence, and the overlapped portion of the External memory will in not be accessed. However, on XA implementations that provide an External Address ($\overline{\text{EA}}$) hardware input, setting EA low will cause external program memory to be used.

3.5.3 Access

Code memory is intended to contain executable XA instructions. The XA architecture supports storing constant data in Code Memory and provides special access modes for retrieving this information. Constant data is implicitly stored within the instruction of many data manipulation instructions when immediate operands are specified.

It is possible, and a common practice, to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such an architecture, implemented with external logic, code memory is logically read-only memory that is writable when accessed as external data memory. No such overlap is possible for internal code memory.

MOVC addressing in Code Memory

A special instruction, MOVC, is defined in the XA for accessing constant data (e.g lookup tables, string constants etc.) stored in code memory. There is a standard form of MOVC that reflects the native XA architecture, and there are two variations that reflect 80C51 compatibility; see Chapter 9 for details of 80C51 compatibility. The standard form of MOVC uses a 16-bit register value as a pointer, appended to either the top 8 bits of the Program Counter (PC) or the Code Segment register (CS) to form a 24-bit address, as shown in Figure 3.12. The source for the upper 8 address bits is determined by the setting of the segment selection bit (0 = PC and 1= CS) in the SSEL register that corresponds to the operand register.

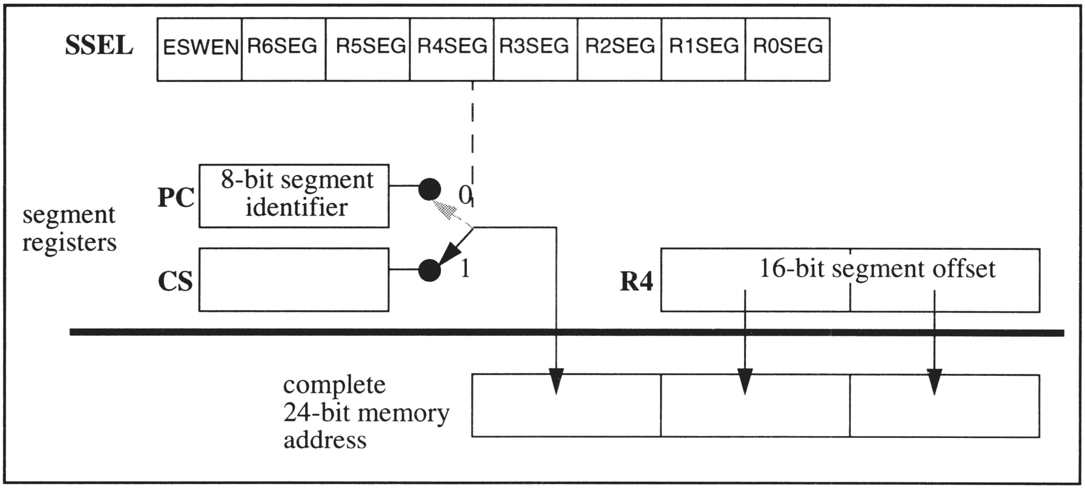


Figure 3.12 MOVC addressing in code memory

3.6 Special Function Registers (SFRs)

Special Function Registers (SFRs) provide a means for programs to access CPU control and status registers, peripheral devices, and I/O ports. The SFR mechanism provides a consistent mechanism for accessing standard portions of the XA core, peripheral functions added to the core within each XA derivative, and external devices as implemented in future derivatives.

Figure 3.13 highlights the core registers that are accessed as SFRs: **PCON**, **SCR**, **SSEL**, **PSWH**, **PSWL**, **CS**, **ES**, **DS**. Communication with these registers is performed via the core's internal SFR bus, which is dedicated for this purpose alone. Communication with peripherals outside the core but on-chip, and with off-chip SFRs is through the SFR Bus Interface. Logically, all these registers are located in the same SFR address space and are all accessed equivalently.

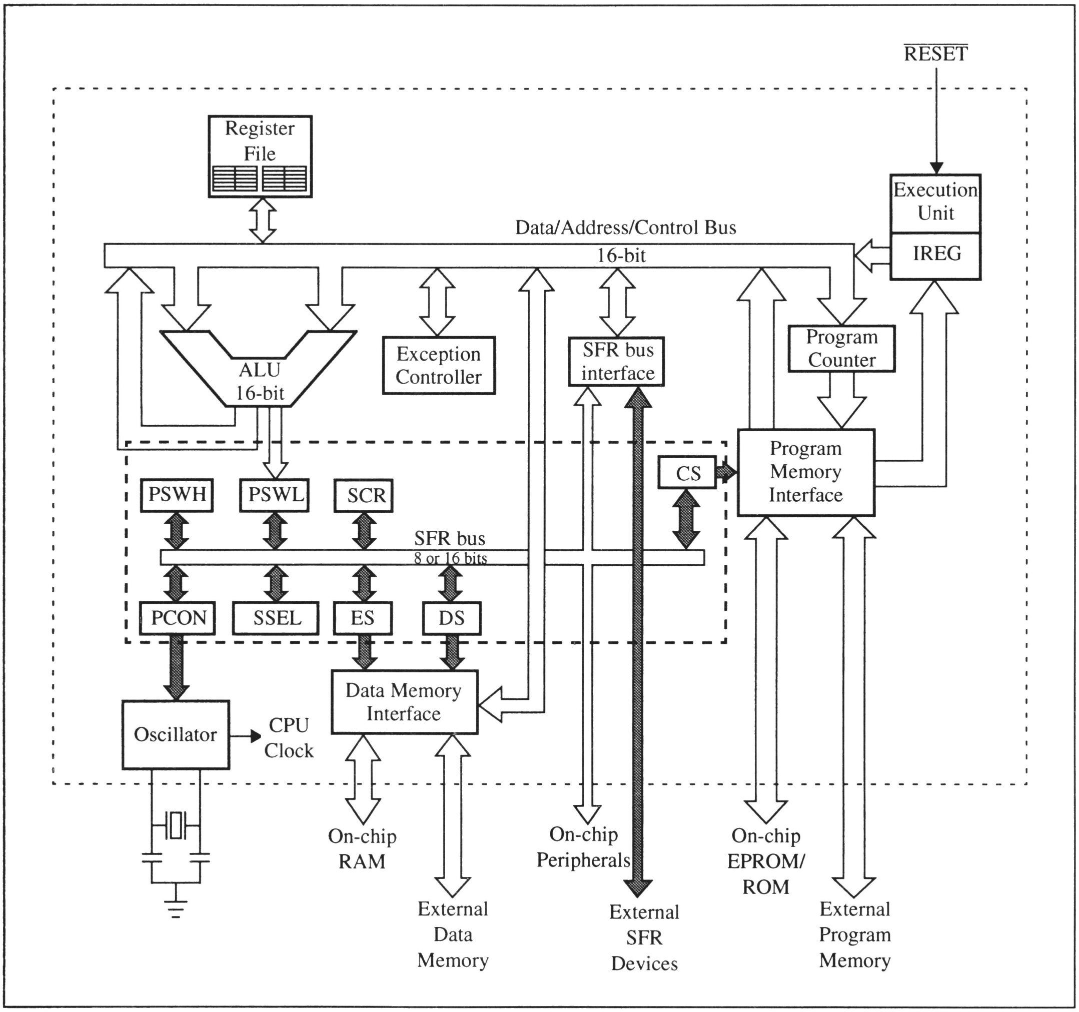

RESET
Register File
Execution Unit
IREG
Data/Address/Control Bus
16-bit
ALU
16-bit
Exception Controller
SFR bus interface
Program Counter
Program Memory Interface
CS
PSWH
PSWL
SCR
SFR bus
8 or 16 bits
PCON
SSEL
ES
DS
Data Memory Interface
Oscillator
CPU Clock
On-chip RAM
External Data Memory
On-chip Peripherals
External SFR Devices
On-chip EPROM/ ROM
External Program Memory

Figure 3.13 XA Core with SFRs highlighted

The SFR address space is 1K bytes (Figure 3.14). The first half of this space (400h through 5FFh) is dedicated to accessing core registers and on-chip peripherals outside the XA core. SFRs assigned addresses in the range 400h through 43Fh are both byte and bit-addressable. The second half (600h through 7FFh) of the SFR space is reserved for providing access to off-chip

SFRs. The off-chip sfr space is provided to allow faster access of off-chip memory mapped I/O devices without having to create a pointer for each access.

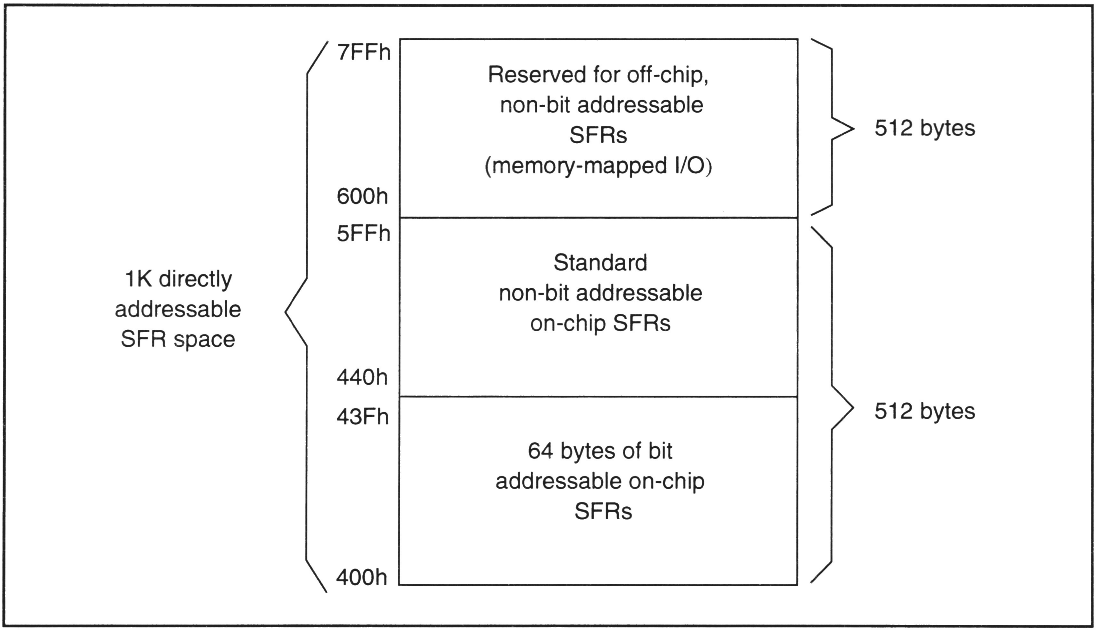

7FFh
Reserved for off-chip, non-bit addressable SFRs (memory-mapped I/O)
512 bytes
600h
5FFh
1K directly addressable SFR space
Standard non-bit addressable on-chip SFRs
440h
43Fh
512 bytes
64 bytes of bit addressable on-chip SFRs
400h

Figure 3.14 SFR address space

Following are some key points to remember when using SFRs:

SFRs should be symbolically addressed. Because SFR assignments may vary from derivative to derivative, it is important to always use symbolic references to SFRs. XA software development tools provide symbolic constants for all SFRs in the form of header/include files and the tools will be updated as new SFRs are added with each added XA derivative.

Verify that your application uses the right header/include files. Although baseline SFRs are likely to retain their addresses in future XA derivatives, this is not guaranteed. SFRs used for optional peripherals may well have different addresses on different derivatives, and the same address on one derivative may access a different peripheral SFR.

Any SFR may be accessed at any time without reference to a pointer or segment. SFR access is independent of any segment register, so SFRs are always accessible with the 10 bit address encoded in instructions accessing SFRs.

SFRs may not be accessed via indirect address. Any time indirection is used, data memory is accessed. If an SFR address is referenced as an indirect address, physical RAM at that address – if it exists– is accessed.

An SFR address is always contained entirely within an instruction. The SFR address is always encoded in the instruction providing the access, and there is no other way of addressing an SFR.

Details of access to external SFRs is determined by derivative implementation. Access to off-chip SFRs is a reserved feature not implemented in the baseline XA. Consult derivative product datasheets for details of external SFR access, e.g., timing.

3.7 Summary of Bit Addressing

Several sections of this chapter have described portions of the XA that are bit-addressable. There are a total of 1024 addressable bits distributed in the XA architecture, chosen to make important data structures immediately accessible via XA bit-processing instructions, specifically, all registers in the register file, R0 through R7 (and R8 through R15 if implemented); directly addressable RAM addresses 20h through 3Fh in the page currently specified by DS, and a portion of the on-chip SFRs. Figure 3.15 summarizes all the bit-addressable portions of the XA.[5]

bit space		overlaps bytes...			
start	end	type	start	end	see
0	0FFh	registers	R0	R15	§3.1
100h	1FFh	direct RAM	20h	3Fh	§3.3
200h	3FFh	on-chip SFRs	400h	43Fh	§3.5

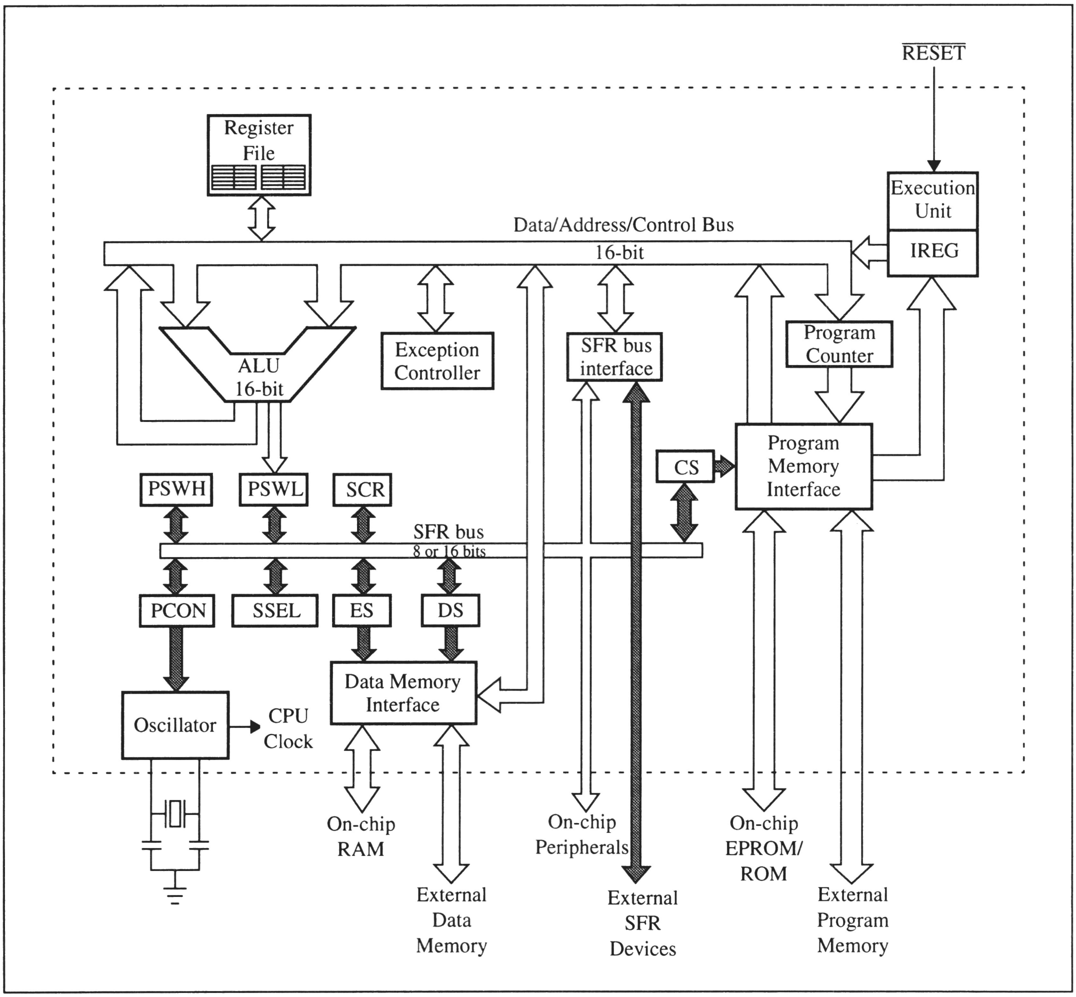
Figure 3.15 Bit addressing summary

4 CPU Organization

This chapter describes the Central Processing Unit (CPU) of the XA Core. The CPU contains all status and control logic for the XA architecture. The XA reset sequence and the system oscillator interface with the CPU, and power control is handled here. The CPU performs interrupt and exception handling. The XA CPU is equipped with special functions to support debugging.

4.1 Introduction

Figure 4.1 is a block diagram of the XA Core.



RESET
Register File
Execution Unit
Data/Address/Control Bus
16-bit
IREG
ALU 16-bit
Exception Controller
SFR bus interface
Program Counter
Program Memory Interface
CS
PSWH
PSWL
SCR
SFR bus
8 or 16 bits
PCON
SSEL
ES
DS
Data Memory Interface
Oscillator
CPU Clock
On-chip RAM
External Data Memory
On-chip Peripherals
External SFR Devices
On-chip EPROM/ ROM
External Program Memory

Figure 4.1 The XA Core

Here is an overview of core elements: The XA Core oscillator provides a basic system clock. Timing and control logic are initialized by an external reset signal; once initialized, this logic provides internal and external timing for program and data memory access. This logic supervises loading the Program Counter and storing instructions fetched by the Program Memory Interface into the Instruction Register. The timing and control logic sequences data transfers to and from the Data Memory Interface. Under the same control, the ALU performs Arithmetic and Logical operations. The ALU stores status information in the low byte of the Program Status Word (**PSWL**). The on-board register file is used for intermediate storage and contains the current value of the Stack Pointer (**SP**). The high byte of the Program Status Word (**PSWH**) chooses between a privileged System Mode and a restricted User Mode; controls a Trace Mode used for single-step debugging, chooses the active register bank, and records the priority of the currently executing process. The System Configuration Register (**SCR**) is initialized to choose native XA mode execution or an 80C51 family compatibility mode. The Segment Selection Register (**SSL**) controls the use of the Code Segment (**CS**), Data Segment (**DS**), and the Extra Segment (**ES**) registers. The XA Core architecture supports interfaces to on- and off-chip RAM, ROM/EPROM, and Special Function Registers (SFRs).

This chapter describes all these core elements in detail.

4.2 Program Status Word

The Program Status Word (**PSW**) is a two-byte SFR register that is a focal point of XA operations. The least significant byte contains the CPU status flags, which generally reflect the result of each XA instruction execution. This byte is readable and writable by programs running in both User and System modes.

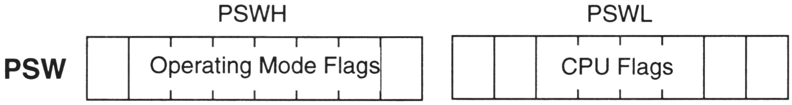


Figure 4.2 XA PSW

The most significant byte of **PSW** is written by programs to set important XA operating modes and parameters: system/user mode, trace mode, register bank select bits, and task execution priority. **PSWH** is readable by any process but only the register select bits may be modified by User mode code. All of the flags may be modified by code running in System Mode.

It should be noted that the XA includes a special SFR that mimics the original 80C51 PSW register. This register, called PSW51, allows complete compatibility with 80C51 code that manipulates bits in the PSW. See Chapter 9 for details of 80C51 compatibility.

4.2.1 CPU Status Flags

The PSW CPU flags (Figure 4.3) signify Carry, Auxiliary Carry, Overflow, Negative, and Zero. Some instructions affect all these flags, others only some of them, and a few XA instructions have no effect on the PSW status flags. In general, these flags are read by programs in order to make logical decisions about program flow. Chapter 6 describes comprehensively how CPU Status Flags are affected by each instruction type. Consult reference pages in Chapter 6 for details about how individual instructions affect the PSW Status Flags.

PSWL	C	AC	-	-	-	V	N	Z

Figure 4.3 PSW CPU status flags

C, the Carry Flag, generally reflects the results of arithmetic and logical operations. It contains the carry out of the most significant bit of an arithmetic operation, if any, for the instructions ADD, ADDC, CMP, CJNE, DA, SUB, and SUBB.The carry flag is also used as an intermediate bit for shift and rotate instructions ASR, LSL, LSR, RLC, and RRC.

The multiply and divide instructions (MUL16, MULU8, MULU16, DIV16, DIV32, DIVU8, DIVU16, and DIVU32) unconditionally clear the carry flag.

AC, the auxiliary carry flag, is updated to reflect the result of arithmetic instructions ADD, ADDC, CMP, SUB, and SUBB with the carry out of the least significant nibble of the ALU. This flag is used primarily to support BCD arithmetic using the decimal adjust instruction (DA).

V is the overflow flag. It is set by a twos complement arithmetic overflow condition during the arithmetic instructions ADD, ADDC, CMP, NEG, SUB, and SUBB.

V is also set when the result of a divide instruction (DIV16, DIV32, DIVU8, DIVU16, DIVU32) exceeds the size of the specified destination register and when a divide-by-zero has occurred. For multiply instructions (MUL16, MULU8, MULU16) this flag is set when the result of a multiply instruction exceeds the source operand size. In this case "overflow" provides an indication to the program that the result is a larger data type than the source, such as a long integer product resulting from the multiply of two integers).

N reflects the twos complement sign (the high-order or "negative" bit) of the result of arithmetic operations and the value transferred by data moves. This flag is unaffected by PUSH and POP instructions.

Z ("zero") reflects the value of the result of arithmetic operations and the value transferred by data moves. This flag is set if the result or value is zero, otherwise it is cleared. The flag is unaffected by PUSH, POP, and XCH instructions.

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.2.2 Operating Mode Flags

The PSW operating mode flags (Figure 4.4) set several aspects of the XA operating mode. All of the flags in the upper byte of the PSW (PSWH) except the bits RS1 and RS0 may be modified only by code running in system mode.

PSWH	SM	TM	RS1	RS0	IM3	IM2	IM1	IM0

Figure 4.4 PSW operating mode flags

The System Mode bit, **SM**, when asserted, allows the currently running program full System Mode access to all XA registers, instructions, and memories. (For example, most of PSWH can only be modified when **SM** is asserted.) When this bit is cleared, the XA is running in User Mode and some privileges are denied to the currently running program.

The Trace Mode bit, **TM**, when set to 1, enables the built-in XA debugging facilities described in section 4.9. When **TM** is cleared, the XA debugging features are disabled.

The bits **RS1** and **RS0** identify one of the four banks of word registers R0 through R3 as the active register set. The other three banks are not accessible as registers (but also see the Compatibility Mode description in the System Configuration Register section).

The 4 bits **IM3** through **IM0** (Interrupt Mask bits) identify the execution priority of the current executing program. The event interrupt controller compares the setting of the IM bits to the priority of any pending interrupts to decide whether to initiate an interrupt sequence. The value 0 in the IM bits indicates the lowest priority, or fully interruptible code. The value 15 (or F hexadecimal) indicates the highest priority, not interruptible by event interrupts. Note that priority 15 does not inhibit servicing of exception interrupts or NMI.

The value of the IM bits may be written only by code operating in the system mode. Their value may be read by interrupt handler code to implement software-based interrupt priorities. Note that simply writing a new value to the interrupt mask bits can sometimes cause what is called a priority inversion, that is, the currently executing code may have a lower priority than previously interrupted code. The Software Interrupt mechanism is included on some XA derivatives specifically to avoid priority inversion in complex systems. Refer to the section on Software Interrupts for details.

4.2.3 Program Writes to PSW

The bytes comprising the PSW, namely PSWH and PSWL, are accessible as SFRs, and there is a potential ambiguity when a write to the PSW is performed by an instruction whose execution also modifies one or more PSW bits. The XA resolves this by giving full precedence to explicit writes to the PSW.

For example, executing

```
MOV R0,#0081h
```

sets PSW bit **N** to 1, since the byte value transferred is a twos complement negative number. However, executing

```
MOVE PSWL, #81h
```

will set PSW bits **C** and **Z** and leave bit **N** cleared, since the value explicitly written to PSW takes precedence.

This precedence rule suppresses *all* PSW flag updates. When a value is written to the PSW, for example when executing

```
OR PSWH, #30
```

the contents of PSWL are unaffected.

4.2.4 PSW Initialization

As described below, at XA reset, the initial PSW value is loaded from the reset vector located at program memory address 0. Philips recommends that the PSW initialization value in the reset vector sets **IM3** through **IM0** to all 1's so that XA initialization is marked as the highest priority process (and therefore cannot be interrupted except by an exception or NMI). At the conclusion of the initialization code, the execution priority is typically reduced, often to 0, to allow all other tasks to run. It is also recommended that the reset vector set the **SM** bit to 1, so that execution begins in System Mode.

4.3 System Configuration Register

The System Configuration Register (**SCR**), described in Figure 4.5, sets XA global operating mode. **SCR** is intended to be written once during system start-up and left alone thereafter. Four bits are currently defined:

SCR	-	-	-	-	PT1	PT0	CM	PZ

Figure 4.5 System Configuration Register (SCR)

PZ set to 0 (the default) puts the XA in the Large-Memory mode that uses full 24-bit XA addressing. When **PZ** = 1 the XA uses a small-memory "Page 0" mode that uses 16 bit addresses. The intent of Page 0 mode is to save stack space and improve interrupt latency in systems with less than 64K bytes of code and data memory. See the following sections for details.

CM chooses between standard "native" mode XA operation and 80C51 compatibility mode. When CM is cleared, the XA operates as described in the first 8 chapters of this manual. When CM is set, the XA operates as described in Chapter 9.

PT1 and **PT0** select a submultiple of the oscillator clock as a Peripheral Timing clock source, in particular for timers but possibly for other peripherals in XA derivatives. Here are the values for these bits and the resulting clock frequency:

PT1	PT0	Peripheral Clock
0	0	oscillator/4
0	1	oscillator/16
1	0	oscillator/64
1	1	reserved

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.3.1 XA Large-Memory Model Description

When the default XA operation is chosen via the **SCR** (**CM** = 0 and **PZ** = 0), all addresses are maintained by the core as 24 bit values, providing a full 16 MByte address space. On a specific XA derivative, fewer than 24 bits may be available at the external bus interface. All 24 address bits are pushed on the stack during calls and interrupts and 24 bits are popped by RETs and RETIs.

4.3.2 XA Page 0 Memory Model Description

When XA Page 0 mode is chosen, only 16 address bits are maintained by the XA core. This operating mode supports XA applications for which a 64K byte address space is sufficient. The external memory interface port used for the upper 8 address bits, if present, is available for other uses. A single 16-bit word is pushed on the stack during calls and interrupts and 16 bits are, in turn popped by RETs and RETIs. Using Page 0 mode when only a small memory model is needed saves stack space and speeds up address PUSH and POP operations on the stack.

Switching into or out of Page 0 mode after the original initialization is not recommended. First, switching into Page 0 mode can only be done by code running on Page 0, since the code address will be truncated to 16-bits as soon as Page 0 mode takes effect. Instructions already in the XA pre-fetch queue would have been fetched prior to Page 0 mode taking effect. Any addresses that may have been pushed onto the stack previously also become invalid when Page 0 mode is changed. Thus Page 0 mode could not be changed while in an interrupt service routine, or any subroutine.

4.4 Reset

The term “reset” refers specifically to the hardware input required when power is first applied to the XA device, and generally to the sequence of initialization that follows a hardware reset, which may occur at any time. The term also refers to the effect of the RESET instruction (see Chapter 6); in addition, an overflowing Watchdog timer (if this peripheral is present) has an identical effect.

This section describes the XA reset sequence and its implications for user hardware and software.

4.4.1 Reset Sequence Overview

A specific hardware reset sequence must be initiated by external hardware when the XA device is powered-up, before execution of a program may begin. If a proper reset at power up is not done, the XA may fail wholly or in part. The XA reset sequence includes the following sequential components:

- Reset signal generated by external hardware
- Internal Reset Sequence occurs
- $\overline{\text{RST}}$ line goes high
- External bus width and memory configuration determined
- Reset exception interrupt generated
- Startup Code executed

Figure 4.6 illustrates this process.

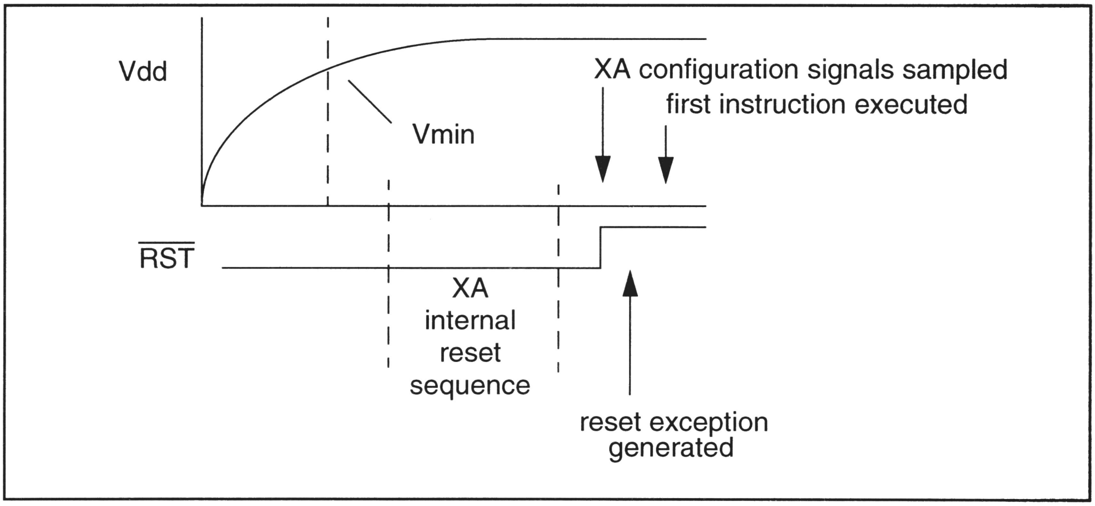

Vdd
Vmin
XA configuration signals sampled
first instruction executed
RST
XA
internal
reset
sequence
reset exception
generated

Figure 4.6 XA power-up sequence

4.4.2 Power-up Reset

This section describes the reset sequence for powering up an XA device.

The XA $\overline{\text{RST}}$ input must be held low for a minimum reset period after Vdd has been applied to the XA device and has stabilized within specifications. The minimum reset period for a typical system with a reasonably fast power supply ramp-up time is 10 milliseconds. This reset period provides sufficient time for the XA oscillator to start and stabilize and for the CPU to detect the reset condition. At this point, the CPU initiates an internal reset sequence. $\overline{\text{RST}}$ must continue to be low for a sufficient time for the internal reset sequence to complete.

4.4.3 Internal Reset Sequence

The XA internal reset sequence occurs after power-up or any time a sufficiently long reset pulse is applied to the $\overline{\text{RST}}$ input while the XA is operating. This sequence requires a minimum of a 10 microseconds (or 10 clocks, whichever is greater) to complete, and $\overline{\text{RST}}$ must remain low for at least this long.

The internal reset sequence does the following:
- Writes a 00 to most core and many peripheral SFRs. Other values are written to some peripheral SFRs. Consult the data sheet of a specific device for details.
- Sets **CS**, **DS**, and **ES** to 0.
- Sets **SSEL** = 0, i.e., sets all accesses through DS.
- Sets all registers in the Register File to 0.
- Sets the user and the system stack pointers (**USP** and **SSP**) to 0100h.
- Clears SCR bit **PZ**, i.e., 24-bit memory addresses will be used by default.
- Clears SCR bit **CM**, i.e., starts execution in XA Native Mode.
- Clears IE bit **EA**, disabling all maskable interrupts.

Note that the internal reset sequence does not initialize internal or external RAM. Note also that the contents of **PSW** at this point is not important, as it will immediately be replaced as described further below.

The effect of the internal reset sequence on components outside the XA core depends on the peripheral complement and configuration of the specific XA derivative. In general, the internal reset sequence has the following effects:

- Sets all port pins to inputs (quasi-bidirectional output configuration with port value = FF hex)
- Clears most SFRs to 0
- Initializes most other SFRs to appropriate non-zero values

Note that serial port buffers, PCA capture registers, and WatchDog feed registers (if present) are unaffected. Consult the XA derivative data sheet for more information.

After the XA internal reset sequence has been completed, the device is quiescent until the $\overline{\text{RST}}$ line goes high.

4.4.4 XA Configuration at Reset

As the $\overline{\text{RST}}$ line goes high, the value on two input pins is sampled to determine the XA memory and bus configuration. The $\overline{\text{EA}}$ and BUSW pins (if present on a specific XA derivative) have special function during the reset sequence, to allow external hardware to determine the use of internal or external program memory, and to select the default external bus width.

Immediately after the $\overline{\text{RST}}$ line goes high, the CPU triggers a reset exception interrupt, as described in the next section.

Selecting Internal or External Program Memory

The XA is capable of reading instructions from internal or external memory, both of which may be present. The XA $\overline{\text{EA}}$ input pin determines whether internal or external program memory will be used. The $\overline{\text{EA}}$ pin is sampled on the rising edge of the $\overline{\text{RST}}$ pulse. If $\overline{\text{EA}} = 0$, the XA will operate out of external program memory, otherwise it will use internal code memory. The selection of external or internal code memory is fixed until the next time $\overline{\text{RST}}$ is asserted and released; until then all code fetches will access the selected code memory.

The XA cannot detect inconsistencies between the setting detected on the $\overline{\text{EA}}$ input and the hardware memory configuration. For example, setting $\overline{\text{EA}} = 1$ on a ROMless XA variant will cause the XA to attempt to execute internal code memory, which is undefined on a ROMless device, typically resulting in a system failure.

Selecting External Bus Width

The XA is capable of accessing an 8 or 16 bit external data bus. The BUSW pin tells the XA the external data bus configuration. The BUSW pin is sampled on the rising edge of the $\overline{\text{RST}}$ pulse. If BUSW is low, the XA operates its external bus interface in 8 bit mode, otherwise, the XA uses 16 bit bus operation. The bus width may also be set under software control on derivatives equipped with the **BCR** ("Bus Configuration Register") SFR.

After $\overline{\text{RST}}$ is released, the BUSW pin may be used an alternate function on some XA derivatives. Consult derivative data sheets for exact pinouts and details of how pins such as these may be shared to keep package size small.

4.4.5 The Reset Exception Interrupt

Immediately after the $\overline{RST}$ line goes high, the CPU generates a Reset Exception Interrupt. As a result, the initial PSW and address of the first instruction (the "start-up code") is fetched from the reset vector in code memory at location 0. Here's an example in generalized assembler format of the setup for the Reset Exception:

```
          code_seg                ; establish code segment
          org  0h                 ; start at address 0

; reset_vector
          dw   initial_PSW        ; define a word constant
          dw   startup_code       ; define a word constant

          org  120h               ; move to address 120h
                                  ; (above vector table)

startup_code:
          ...                     ; put startup code here
```

The initial value of **PSWL** set in the Reset Vector is generally of no special system-wide importance and may be set to zero or some other value to meet special needs of the XA application. The initial **PSWH** value sets the stage for system software initialization and its value requires more attention. Here's an example set of declarations that create the recommended initial value of **PSWH**:

```
system_mode     equ  8000h
max_priority    equ  0F00h
initial_PSW     equ  system_mode + max_priority
```

It is generally appropriate to initialize the XA in System Mode so that the start-up code has unrestricted access to the entire architecture. This is done by using a initial value that sets the PSWH bit **SM**.

Philips recommends initializing the execution priority of the start-up code to the highest possible value of 15 (that is, IM0 through IM3 to all ones) so that the start-up code is recognizable as the highest priority process. As described above, the hardware initialization sequence turns off all possible interrupts, so the only potential interrupting process would arise from a non-maskable interrupt (NMI). It is generally a good idea to prevent NMI generation with a hardware lock-out until XA start-up procedures are completed.

The **PSWH** initialization value given in this example sets System Mode (**SM**), selects register bank 0 (any register bank could be used) and clears **TM** so that Trace Mode is inactive.

4.4.6 Startup Code

Philips recommends that the first instruction of start-up code set the value of the System Configuration Register (**SCR**), described in section 4.3, to reflect the system architecture.

The next recommended step is explicitly initializing the stack pointers. The default values (section 4.7) are usually insufficient for application needs.

The start-up code sequence may be concluded by a simple branch or jump to application code. A RETI may not be used at the conclusion of a Reset Exception Interrupt handler (which causes the start-up code to run) because a reset initializes the SP and does not leave an interrupt stack frame.

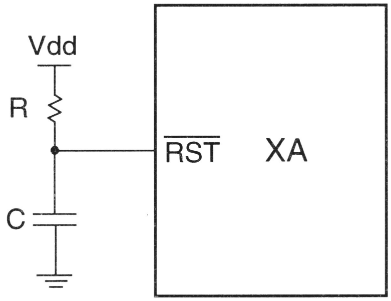
4.4.7 Reset Interactions with XA Subsystems

The following describes how the reset process interacts with some key subsystems:

- Trace Exception. The trace exception is aborted by an external reset; see section 4.9.
- WatchDog. In XA derivatives equipped with a WatchDog timer feature, $\overline{\text{RST}}$ will be asserted for at least a derivative-defined number of cycles. The $\overline{\text{RST}}$ pin is driven low for this period.
- Resets while in Idle Mode. Since the XA oscillator is running in Idle Mode, the $\overline{\text{RST}}$ input must be kept low for only 10 microseconds (or 10 clocks, whichever is greater) to achieve a complete reset.
- Resets while in Power-Down Mode. The XA oscillator is stopped in Power-Down mode, so the $\overline{\text{RST}}$ input must be low for at least 10 milliseconds.

4.4.8 An External Reset Circuit

The $\overline{\text{RST}}$ pin is a high-impedance Schmitt trigger input pin. For applications that have no special start-up requirements, it is practical to generate a reset period known to be much longer than that required by the power supply rise time and by the XA under all foreseeable conditions. One simple way to build a reset circuit is illustrated in Figure 4.7.



Some typical values for R and C:

R = 100K, C = 1.0μF

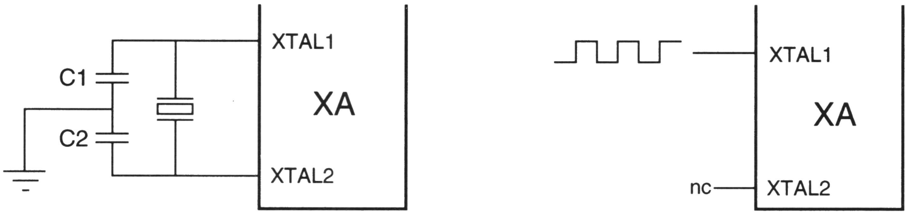
R = 1M, C = 0.1μF

(assuming that the Vdd rise time is 1 millisecond or less)

Figure 4.7 An external reset circuit

4.5 Oscillator

The XA contains an on-chip oscillator which may be used as the clock source for the XA CPU, or an external clock source may be used. A quartz crystal or ceramic resonator may be connected as shown in Figure 4.8a to use the internal oscillator. To use an external clock, connect the source to pin XTAL1 and leave pin XTAL2 open, as shown in Figure 4.8b.



a) using the on-chip oscillator

b) using an external clock

Figure 4.8 XA clock sources

The on-chip oscillator of the XA consists of a single stage linear inverter intended for use as a positive reactance oscillator. In this application, the crystal is operated in its fundamental response mode as an inductive reactance in parallel resonance with capacitance external to the crystal.

A quartz crystal or ceramic resonator is connected between the XTAL1 and XTAL2 pins, capacitors ar connected from both pins to ground. In the case of a quartz crystal, a parallel resonant crystal must be used in order to obtain reliable operation. The capacitor values used in the oscillator circuit should normally be those recommended by the crystal or resonator manufacturer. For crystals, the values may generally be from 18 to 24 pF for frequencies above 25 MHz and 28 to 34 pF for lower frequencies. Too large or too small capacitor values may prevent oscillator start-up or adversely affect oscillator start-up time.

4.6 Power Control

The XA CPU implements two modes of reduced power consumption: Idle mode, for moderate power savings, and Power-Down mode. Power-Down reduces XA consumption to a bare minimum. These modes are initiated by writing SFR **PCON**, as illustrated in Figure 4.9.

PCON	-	-	-	-	-	-	PD	IDL

Figure 4.9 PCON

Idle Mode is activated by setting the PCON bit **IDL**. This stops CPU execution while leaving the oscillator and some peripherals running.

Power-Down mode is activated set by setting the PCON bit **PD**. This shuts down the XA entirely, stopping the oscillator.

The reset values of **IDL** and **PD** are 0. If a 1 is written to both bits simultaneously, **PD** takes precedence and the XA goes into Power-Down mode.

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.6.1 Idle Mode

Idle mode stops program execution while leaving the oscillator and selected peripherals active. This greatly reduces XA power consumption. Those peripheral functions may cause interrupts (if the interrupt is enabled) that will cause the processor to resume execution where it was stopped.

In the Idle mode, the port pins retains their logical states from their pre-idle mode. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). ALE and $\overline{\text{PSEN}}$ are held in their respective non-asserted states. When Idle is exited normally (via an active interrupt), port values and configurations will remain in their original state.

4.6.2 Power-Down Mode

Power-Down mode stops program execution and shuts down the on-chip oscillator. This stops all XA activity. The contents of internal registers, SFRs and internal RAM are preserved. Further power savings may be gained by reducing XA Vdd to the RAM retention voltage in Power Down mode; see the device data sheet for the applicable Vdd value. The processor may be re-activated by the assertion of $\overline{\text{RST}}$ or by assertion of one of an external interrupt, if enabled. When the processor is re-activated, the oscillator will be restarted and program execution will resume where it left off.

In Power-Down mode, the ALE and $\overline{\text{PSEN}}$ outputs are held in their respective non-asserted states. The port pins output the values held by their respective SFRs. Thus, port pins that are not configured to be part of an external bus retain their state. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). If Power-Down mode is exited via Reset, all port values and configurations will be set to the default Reset state.

If Power-Down mode is exited via an external interrupt, port values and configurations will remain in their original state. Since the XA oscillator is stopped when the XA leaves Power-Down mode via an interrupt, time must be allowed for the oscillator to re-start. Rather than force the external logic asserting the interrupt to remain active during the oscillator start-up time, the

XA implements its own timer to insure proper wake-up. This timer counts 9,892 oscillator clocks before allowing the XA to resume program execution, thus insuring that the oscillator is running and stable at that time.

Note that if an external oscillator is used, power supply current reduction in the Power-Down mode is reduced from what would be obtained when using the XA on-chip oscillator. In this case, full power savings may be gained by turning off the external clock source or stopping it from reaching the XTAL1 pin of the XA. If the clock source may be turned off, it may be advantageous to use Idle mode rather than Power-Down mode, to allow more ways of terminating the power reduction mode and to avoid the 9,892 clock waiting period for exiting Power-Down mode.

4.7 XA Stacks

The XA stacks are word-aligned LIFO data structures that grow downward in data memory, from high to low address. This and some other details of the XA stack implementation differ from 80C51 stack operation. Refer to the chapter on 8051 compatibility for a detailed discussion of this topic.

The XA implements two distinct stacks, one for User Mode and one for System Mode. The User Stack may be placed anywhere in data memory, while the System Stack must be located in the first 64K bytes, i.e., segment 0.

4.7.1 The Stack Pointers

The XA has two stacks, the system stack and the user stack. Each stack has an associated stack pointer, the System Stack Pointer (SSP) and the User Stack Pointer (USP), respectively. Only one of these stacks is active at a given time. The current stack pointer at any instant (which may be the SSP or the USP) appears as word register R7 in the register file; the other stack pointer will not be visible. The value of the PSW bit **SM** determines which stack is active (and whose stack pointer therefore appears as R7) . In User Mode (**SM** = 0), R7 contains the User Stack Pointer. In System Mode (**SM** =1), R7 contains the System Stack Pointer. The XA automatically switches SSP and USP values when the operating mode is changed. Note that the terms “USP” and “SSP” are logical terms, denoting the value of R7 in each mode.

Segments and Protection

The User stack is always addressed relative to the current data segment (DS) value. This is consistent with each user task being associated with a specific data segment. Moreover, code running in User Mode cannot modify **DS**, so there is no possibility of changing the segment in which the stack resides within the User context. The System Stack must always be located in segment 0, that is, the first 64K of data memory.

4.7.2 PUSH and POP

The PUSH operation is illustrated by Figure 4.10. The stack pointer always points to an existing data item at the top of the stack, and is decremented by 2 prior to writing data.

The POP operation copies the data at the top of the stack and then adds two to the stack pointer, as follows shown in Figure 4.11.

All stack pushes and pops occur in word multiples. If a byte quantity is pushed on the stack it is stored as the least significant byte of a word and the high byte is left unwritten;
see Figure 4.12. A POP to a byte register removes a word from the stack and the byte register receives the least significant 8 bits of the word, as shown in Figure 4.13.

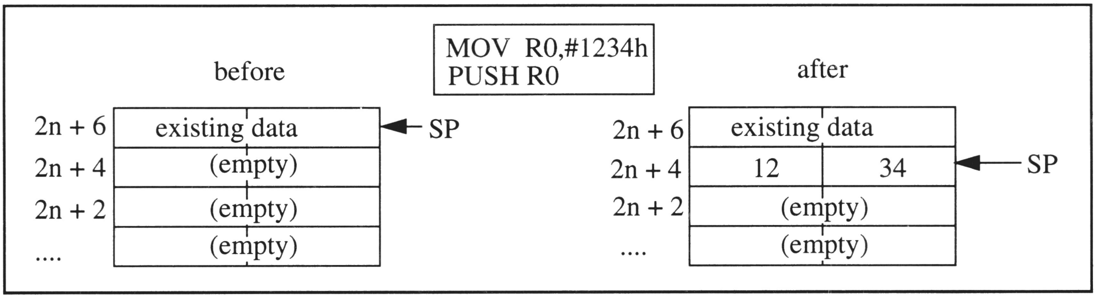

before
MOV R0,#1234h
PUSH R0
after
2n + 6
2n + 4
2n + 2
....
existing data
(empty)
(empty)
(empty)
SP
2n + 6
2n + 4
2n + 2
....
existing data
12
34
(empty)
(empty)
SP

Figure 4.10 PUSH operation

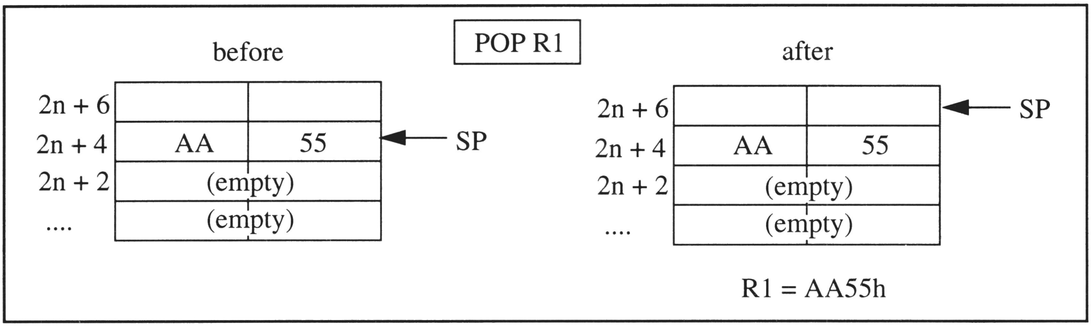

before
POP R1
after
2n + 6
2n + 4
2n + 2
....
AA
55
(empty)
(empty)
SP
2n + 6
2n + 4
2n + 2
....
AA
55
(empty)
(empty)
SP
R1 = AA55h

Figure 4.11 POP operation

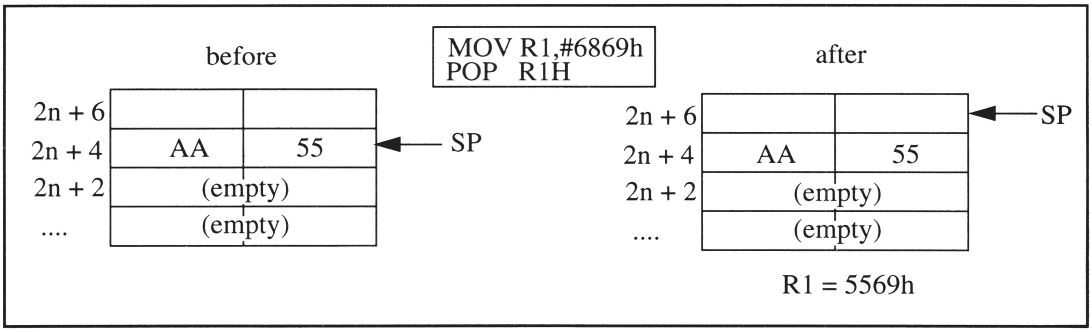

before
MOV R1,#6869h
POP R1H
after
2n + 6
2n + 4
2n + 2
....
AA
55
(empty)
(empty)
SP
2n + 6
2n + 4
2n + 2
....
AA
55
(empty)
(empty)
SP
R1 = 5569h

Figure 4.12 POP a byte

The stack should always be word-aligned. If R7 is modified to an odd value, the offending LSB of the stack pointer is ignored and the word at the next-lower even address is accessed.

Note that neither PUSH or POP operations have any effect on the PSW flags.

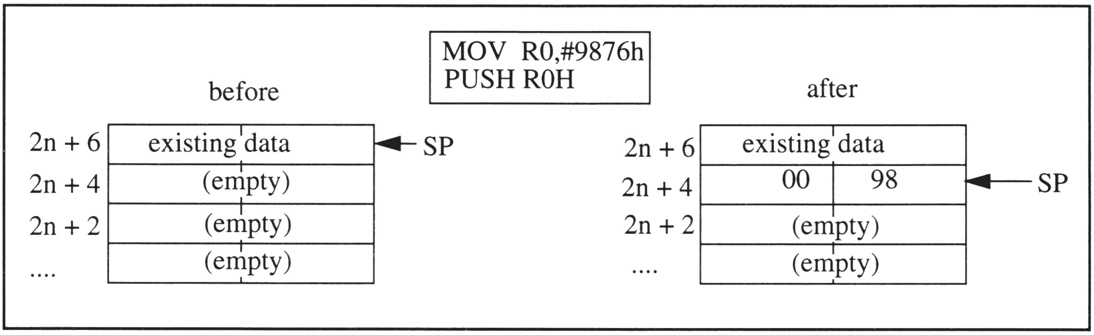


Figure 4.13 PUSH a byte

4.7.3 Stack-Based Addressing

Stack-based data addressing is fully supported by the XA. R0 through R7 may be used in all indexed address modes; the stack pointer in R7 is equally valid as an index.
Figure 4.14 illustrates an example of stack-based addressing.The segment used for stack relative addressing is always the same as for other stack operations (Segment 0 for System mode code and DS for User mode code). Note that the precautions mentioned in section 3.3.4 apply here: when referencing a word quantity, the final (effective) address must be even, otherwise incorrect data will be accessed. This topic is discussed further in the section Stack Pointer Misalignment.

4.7.4 Stack Errors

Special attention is required to avoid problems due to stack overflow, stack underflow, and stack pointer misalignment

Stack Overflow

Stack overflow occurs when too many items are pushed, either explicitly or as the result of interrupts. As items are pushed on to the stack, it may grow downward past the memory allocated to it. It is not always possible for programs to detect stack overflow, so the XA triggers a Stack Overflow Exception Interrupt whenever the value of the *current* stack pointer (SSP or USP) decrements from 80h to 7Eh (simply setting SP to a value lower than 80h would NOT cause a stack overflow). This value was chosen so that stack space sufficient to handle a stack overflow exception interrupt is always guaranteed, as follows:

The 80h limit leaves 64 bytes available for stack overflow processing. A worst case might be occurs when the Stack Pointer is at 80h and a program executes an 8 word push; this generates a stack overflow. If an NMI occurs at the same time, 3 additional words are pushed. The balance of the 64 bytes on the stack is available for handler processing, which should carefully limit further use of the stack.

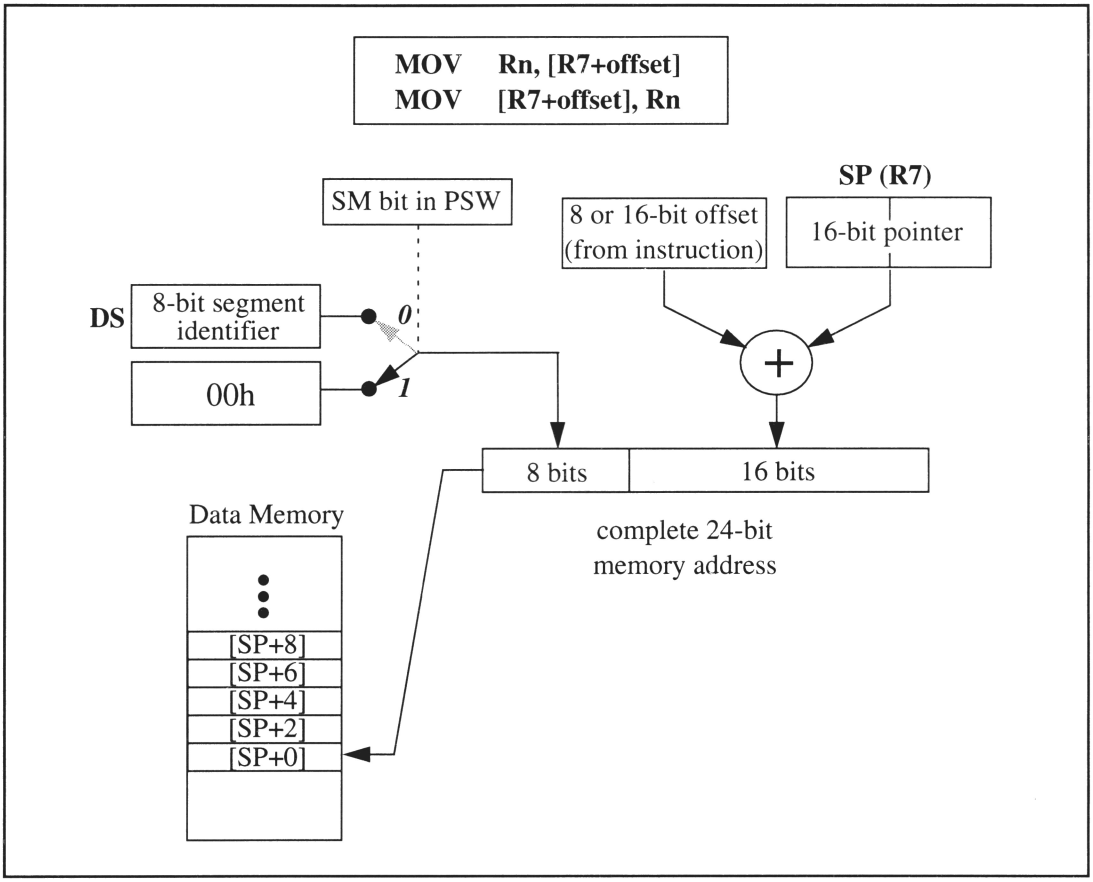

MOV Rn, [R7+offset]
MOV [R7+offset], Rn
SM bit in PSW
SP (R7)
8 or 16-bit offset (from instruction)
16-bit pointer
DS
8-bit segment identifier
0
00h
1
+
8 bits
16 bits
complete 24-bit memory address
Data Memory
[SP+8]
[SP+6]
[SP+4]
[SP+2]
[SP+0]

Figure 4.14 Stack-based addressing

Stack Underflow

Stack underflow occurs when too many items are popped and the stack pointer value becomes greater than its initial value, i.e., the stack top. The XA does not support stack underflow detection.

Stack Pointer Misalignment

Pointer misalignment occurs when a pointer contains an odd value and is used by an instruction to access a word value in memory. The same situation could occur if some program action forced the stack pointer to an odd value. In these cases, the XA ignores the bottom bit of the pointer and continues with a word memory access.

4.7.5 Stack Initialization

At power-on reset, *both* USP and SSP in all XA derivatives are initialized to 100h, that is, one byte past the minimum on-chip XA RAM space (256 bytes). Since SP is pre-decremented, the first PUSH operation will store a word at location FEh and the stack will grow downwards from there.

These default stack pointer start-up values overlap the System and User stacks and are applicable only when one of these stacks will never be used.

Since the System stack is used for all exception and interrupt processing, this may not be appropriate in all XA applications. The startup code should normally set new and different values of both USP and SSP.

4.8 XA Interrupts

The XA architecture defines four kinds of interrupts. These are listed below in order of intrinsic priority:

- Exception Interrupts
- Event Interrupts
- Software Interrupts
- Trap Interrupts

Exception interrupts reflect system events of overriding importance. Examples are stack overflow, divide-by-zero, and Non-Maskable Interrupt. Exceptions are always processed immediately as they occur, regardless of the priority of currently executing code.

Event interrupts reflect less critical hardware events, such as a UART needing service or a timer overflow. Event interrupts may be associated with some on-chip device or an external interrupt input. Event interrupts are processed only when their priority is higher than that of currently executing code. Event interrupt priorities are settable by software.

Software interrupts are an extension of event interrupts, but are caused by software setting a request bit in an SFR. Software interrupts are also processed only when their priority is higher than that of currently executing code. Software interrupt priorities are fixed at levels from 1 through 7.

Trap interrupts are processed as part of the execution of a TRAP instruction. So, the interrupt vector is always taken when the instruction is executed.

All forms of interrupts trigger the same sequence: First, a *stack frame* containing the address of the next instruction and then the current value of the PSW is pushed on the System Stack. A vector containing a new PSW value and a new execution address is fetched from code memory. The new PSW value entirely replaces the old, and execution continues at the new address, i.e., at the specific interrupt handler.

The new PSW value may include a new setting of PSW bit **SM**, allowing handler routines to be executed in System or User mode, and a new value of PSW bits **IM3** through **IM0**, reflecting the execution *priority* of the new task. These capabilities are basic to multi-tasking support on the XA. See Chapter 5 for more details.

Returns from all interrupts should in most cases be accomplished by the RETI instruction, which pops the System Stack and continues execution with the restored PSW context. Since RETI executed while in User Mode will result in an exception trap, as described further below, interrupt service routines will normally be executed in System Mode.

The XA architecture contains sophisticated mechanisms for deciding when and if an interrupt sequence actually occurs. As described below, Exception Interrupts are always serviced as soon as they are triggered. Event Interrupts are deferred until their execution priority is higher than that of the currently executing code. For both exception and event interrupts, there is a systematic way of handling multiple simultaneous interrupts. Software and trap interrupts occur only when program instructions generating them are executed so there is no need for conflict resolution.

The Non-Maskable Interrupt requires special consideration. It is generated outside the XA core, and in that respect is an event interrupt. However, it shares many characteristics of exception interrupts, since it is not maskable. Note that NMI, while part of the XA CPU core, may not always be connected to a pin or other event source on all XA derivatives.

4.8.1 Interrupt Type Detailed Descriptions

This section describes the four kinds of interrupts in detail.

Exception Interrupts

Exception interrupts reflect events of overriding importance and are always serviced when they occur. Exceptions currently defined in the XA core include: Reset, Breakpoint, Divide-by-0, Stack overflow, Return from Interrupt (RETI) executed in User Mode, and Trace. Ten additional exception interrupts are reserved. NMI is listed in the table of exception interrupts (Table 4.1) below because NMI is handled by the XA core in same manner as exceptions, and factors into the precedence order of exception processing.

Since exception interrupts are by definition not maskable, they must always be serviced immediately regardless of the priority level of currently executing code, as defined by the IM bits in the PSW. In the unusual case that more than one exception is triggered at the same time, there is a hard-wired *service precedence* ranking. This determines which exception vector is taken first if multiple exceptions occur. In these cases, the exception vector taken *last* may be considered the highest priority, since its code will execute first. Of course, being non-maskable, any exception occurring during execution of the ISR for another exception will still be serviced immediately.

Programmers should be aware of the following when writing exception handlers:

1. Since another exception could interrupt a stack overflow exception handler routine, care should be taken in all exception handler code to minimize the possibility of a destructive stack overflow. Remember that stack overflow exceptions only occur once as the stack crosses the bottom address limit, 80h.

2. The breakpoint (caused by execution of the BKPT instruction, or a hardware breakpoint in an emulation system) and Trace exceptions are intended to be mutually exclusive. In both cases, the handler code will want to know the address in user code where the exception occurred. If a breakpoint occurs during trace mode, or if trace mode is activated during execution of the breakpoint handler code, one of the handlers will see a return address on the stack that points within the other handler code.

Table 4.1: Exception interrupts, vectors, and precedence

Exception Interrupt	Vector Address	Service Precedence
Breakpoint	0004h:0007h	0
Trace	0008h:000Bh	1
Stack Overflow	000Ch:000Fh	2
Divide-by-zero	0010h:0013h	3
User RETI	0014h:0017h	4
<reserved>	0018h - 003Fh	5
NMI	009Ch:009Fh	6
Reset	0000h:0003h	7 (always serviced immediately, aborts other exceptions)

Event Interrupts

Event Interrupts are typically related to on-chip or off-chip peripheral devices and so occur asynchronously with respect to XA core activities. The XA core contains no inherent event interrupt sources, so event interrupts are handled by an interrupt control unit that resides on-chip but outside of the processor core.

On typical XA derivatives, event interrupts will arise from on-chip peripherals and from events detected on interrupt input pins. Event interrupts may be globally enabled/disabled via the **EA** bit in the Interrupt Enable register (IE) and individually masked by specific bits the IE register or its extension. An event interrupt that is enabled is serviced when its execution priority is higher than that of the currently executing code. Each event interrupt is assigned a priority level in the Interrupt Priority register(s). If more than one event interrupt occurs at the same time, the priority setting will determine which one is serviced first. If more than one interrupt is pending at the same level priority, a hardwares precedence scheme is used to choose the first to service. Consult the data sheet for a specific XA derivative for details.

Note that, like all other forms of interrupts, the PSW (including the Interrupt Mask bits) is loaded from the interrupt vector table when an event interrupt is serviced. Thus, the priority at which the interrupt service routine executes could be different than the priority at which the interrupt occurred (since that was determined not by the PSW image in the vector table, but by the Interrupt Priority register setting for that interrupt). Normally, it is advisable to set the execution priority in the interrupt vector to be the same as the Interrupt Priority register setting that will be used in the program.

Software Interrupts

Software Interrupts act just like event interrupts, except that they are caused by software writing to an interrupt request bit in an SFR. The standard implementation of the software interrupt mechanism provides 7 interrupts which are associated with 2 Special Function Registers. One SFR, the software interrupt request register (SWR), contains 7 request bits: one for each software interrupt. The second SFR is an enable register (SWE), containing one enable bit matching each software interrupt request bit.

Software interrupts have fixed interrupt priorities, one each at priorities 1 through 7. These are shown in table 4.2 below. Software Interrupts are defined outside the XA core and may not be present on all XA derivatives; consult the specific XA derivative data sheet for details.

Table 4.2: Software interrupts, vectors, and fixed priorities

Software Interrupt	Vector Address	Fixed Priority
SWI1	0100h:0103h	1
SWI2	0104h:0107h	2
SWI3	0108h:010Bh	3
SWI4	010Ch:010Fh	4
SWI5	0110h:0113h	5
SWI6	0114h:0117h	6
SWI7	0118h:011Bh	7

The primary purpose of the software interrupt mechanism is to provide an organized way in which portions of event interrupt routines may be executed at a lower priority level than the one at which the service routine began. An example of this would be an event Interrupt Service Routine that has been given a very high priority in order to respond quickly to some critical external event. This ISR has a relatively small portion of code that must be executed immediately, and a larger portion of follow-up or “clean-up” code which does not need to be completed right away. Overall system performance may be improved if the lower priority portion of the ISR is actually executed at a lower priority level, allowing other more important interrupts to be serviced.

If the high priority ISR simply lowers its execution priority at the point where it enters the follow-up code, by writing a lower value to the IM bits in the PSW, a situation called “priority inversion” could occur. Priority inversion describes a case where code at a lower priority is executing while a higher priority routine is kept waiting. An example of how this could occur by writing to the IM bits follows, and is illustrated in figure 4.15.

Suppose code is executing at level 0 and is interrupted by an event interrupt that runs at level 10. This is again interrupted by a level 12 interrupt. The level 12 ISR completes a time-critical portion of its code and wants to lower the priority of the remainder of its code (the non-time critical portion) in order to allow more important interrupts to occur. So, it writes to the IM bits, setting the execution priority to 5. The ISR continues executing at level 5 until a level 8 event interrupt occurs. The level 8 ISR runs to completion and returns to the level 5 ISR, which also runs to completion. When the level 5 ISR returns, the previously interrupted level 10 ISR is re-activated and eventually competes.

It can be seen in this example that lower priority ISR code executed and completed while higher priority code was kept waiting on the stack. This is priority inversion.

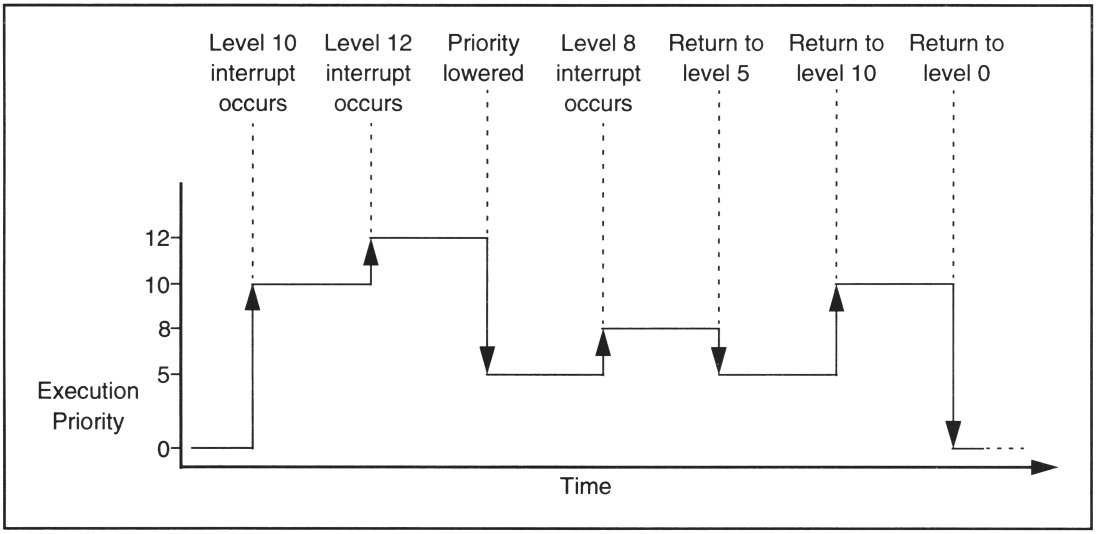


Figure 4.15 Example of priority inversion (see text)

In those cases where it is desirable to alter the priority level of part of an ISR, a software interrupt may be used to accomplish this without risk of priority inversion. The ISR must first be split into 2 pieces: the high priority portion, and the lower priority portion. The high priority portion remains associated with the original interrupt vector. The lower priority portion is associated with the interrupt vector for software interrupt 5. At the completion of the high priority portion of the ISR, the code sets the request bit for software interrupt 5, then returns. the remainder of the ISR, now actually the ISR for software interrupt 5, executes when it becomes the highest priority pending interrupt.

The diagram in figure 4.16 shows the same sequence of events as in the example of priority inversion, except using software interrupt 5 as just described. Note that the code now executes in the correct order (higher priority first).

Trap Interrupts

Trap Interrupts are generated by the TRAP instruction. TRAP 0 through TRAP 15 are defined and may be used as required by applications. Trap Interrupts are intended to support application-

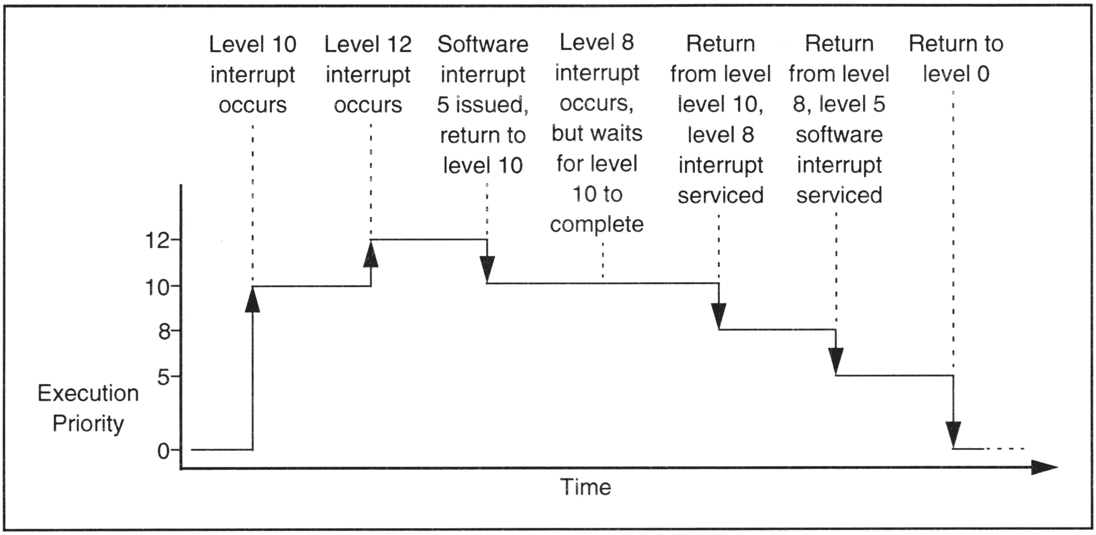


Figure 4.16 Example use of software interrupt (see text)

specific requirements, as a convenient mechanism to enter globally used routines, and to allow transitions between user mode and system mode. A trap interrupt will occur if and only if the instruction is executed, so there is no need for a precedence scheme with respect to simultaneous traps.

See Chapter 6 for a detailed description of the TRAP instruction.

4.8.2 Interrupt Service Data Elements

There are two data elements associated with XA interrupts. The first is the stack frame created when each interrupt is serviced. The second is the interrupt vector table located at the beginning of code memory. Understanding the structure and contents of each is essential to the understanding of how XA interrupts are processed.

Interrupt Stack Frame

A stack frame is generated, always on the System Stack, for each XA interrupt. With one exception, the stack frame is stored for the duration of interrupt service and used to return to and restore the CPU state of the interrupted code. (The exception is an Exception Interrupt triggered by a Reset event. Since Reset re-initializes the stack pointers, no stack frame is preserved. See section 4.4 for details.) The stack frame in the native 24-bit XA operating mode is illustrated in Figure 4.17. Three words are stored on the stack in this case. The first word pushed is the low-order 16 bits of the current PC, i.e., the address of the next instruction to be executed. The next word contains the high-order byte of the current PC. A zero byte is used as a pad. In sum, a complete 24-bit address is stored in the stack frame. The third word contains a copy of the PSW at the instant the interrupt was serviced.

When the XA is operating in Page 0 Mode (SCR bit **PZ** = 1) the stack frame is smaller because, in this mode, only 16 address bits are used throughout the XA. The stack frame in Page 0 Mode is illustrated in Figure 4.18. Obviously it is very important that stack frames of both sizes not be mixed; this is one reason for the admonition in section 4.3 to set the System Configuration Register once during XA initialization and leave it unchanged thereafter.

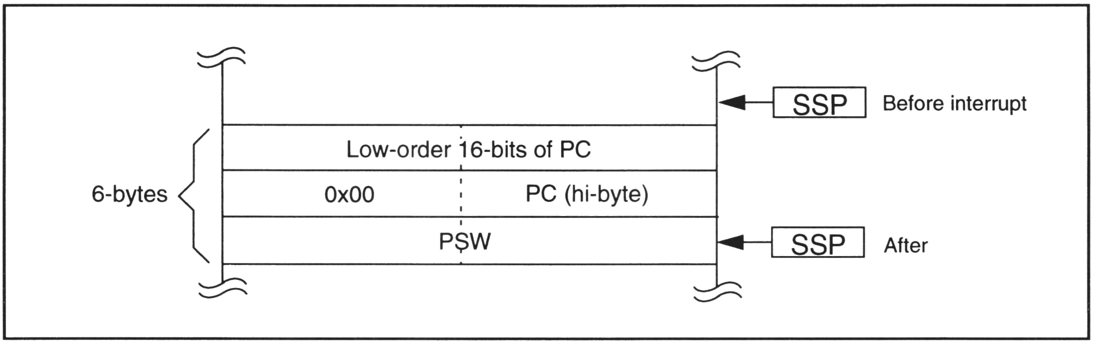


Figure 4.17 Interrupt stack frame (non- page zero mode)

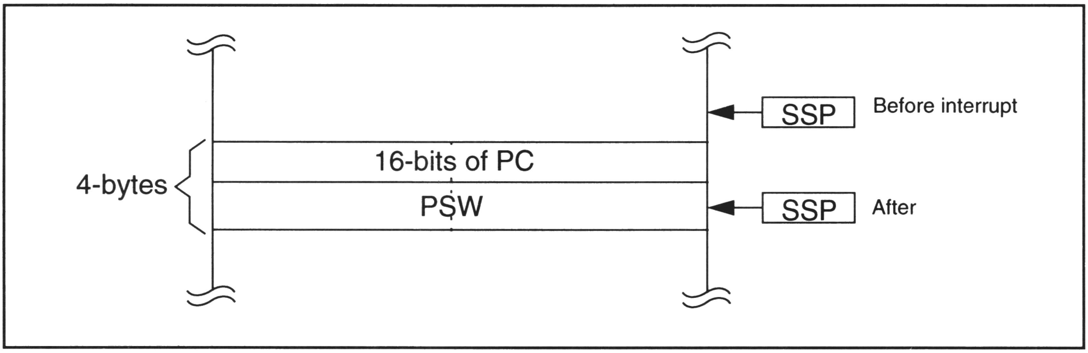


Figure 4.18 Interrupt stack frame (page 0 mode)

Interrupt Vector Table

The XA uses the first 284 bytes of code memory (addresses 0 through 11B hex) for an interrupt vector table. The table may contain up to 71 double-word entries, each corresponding to a particular interrupt event.

The double-word entries each consist of a 16 bit address of an interrupt service routine address and a 16 bit PSW replacement value. Because vector addresses are 16-bit, the first instruction of service routines must be located in the first 64K bytes of XA memory. The first instruction of all service routines must be word-aligned. Key elements of the replacement PSW value are the choice of System or User mode for the service routine, the Register Bank selection, and an Execution Priority setting. For more details on PSW elements, see section 4.2.2.

The first 16 vectors, starting at code memory address 0 are reserved for Exception Interrupt vectors. The second 16 vectors are reserved for Trap Interrupts. The following 32 vectors in the table are reserved for Event Interrupts. The final 7 vectors are used for Software Interrupts. Figure 4.19 illustrates the XA vector table and the structure of each component vector. Of the vectors assigned to Exceptions, 6 are assigned to events specific to the XA CPU and 10 are reserved. All 16 Trap Interrupts may be used freely. Assignments of Event Interrupt vectors are derivative-independent and vary with the peripheral device complement and pinout of each XA derivative.

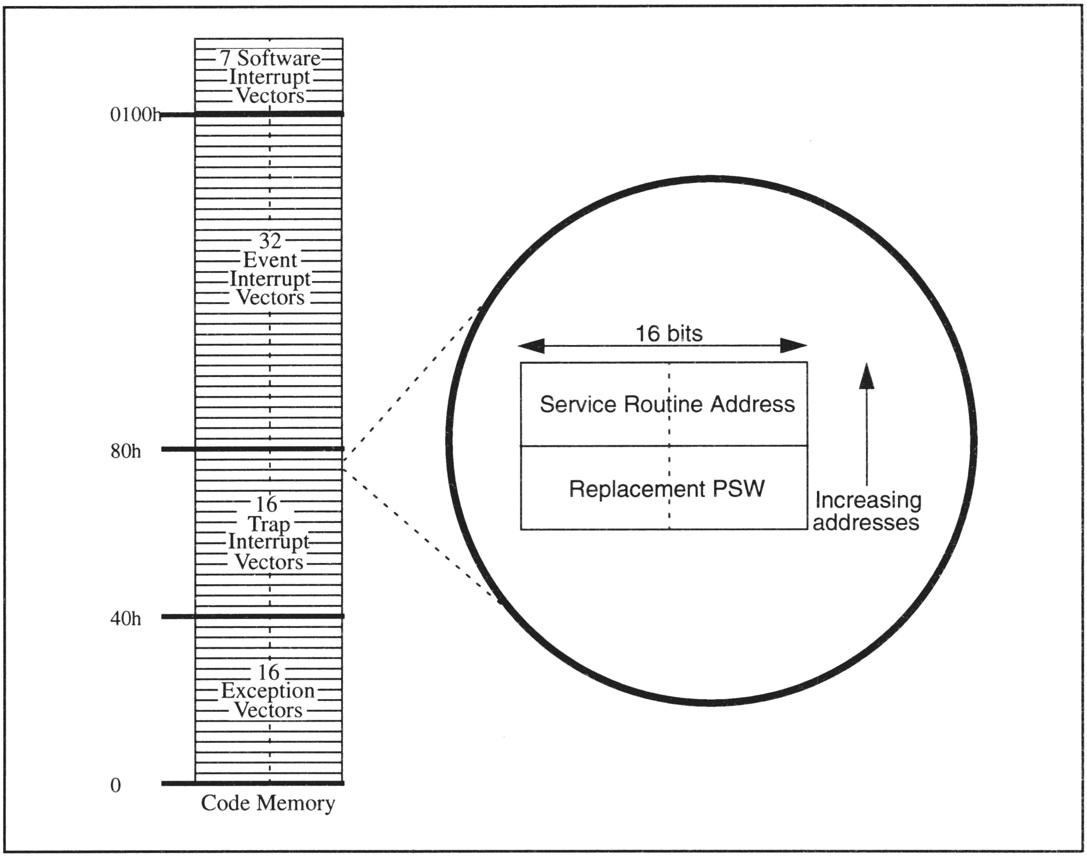

7 Software Interrupt Vectors
0100h
32 Event Interrupt Vectors
80h
16 Trap Interrupt Vectors
40h
16 Exception Vectors
0
Code Memory
16 bits
Service Routine Address
Replacement PSW
Increasing addresses

Figure 4.19 Interrupt vectors

Unused interrupt vector locations should typically be set to point to a "null" service routine (an RETI instruction), rather than be overwritten by executable instructions. This is especially true of the exception interrupts and NMI, since these could conceivably occur in a system where the designer did not expect them. If these vectors are routed to an RETI instruction, the system can essentially ignore the unexpected exception or interrupt condition and continue operation.

4.9 Trace Mode Debugging

The XA has an optional Trace Mode in which a special trace exception is generated at the conclusion of each instruction. Trace Mode supports user-supplied debugger/monitor programs which can single-step through any code, even code in ROM.

4.9.1 Trace Mode Operation

Trace Mode is initiated by asserting **PSW.TM** in the context of the program to be traced.

Using Trace Mode requires a detailed understanding of the XA instruction execution sequence because when and if a trace exception occurs depends on events within the execution sequence of a single instruction. Figure 4.20 illustrates the XA instruction sequence in overview.

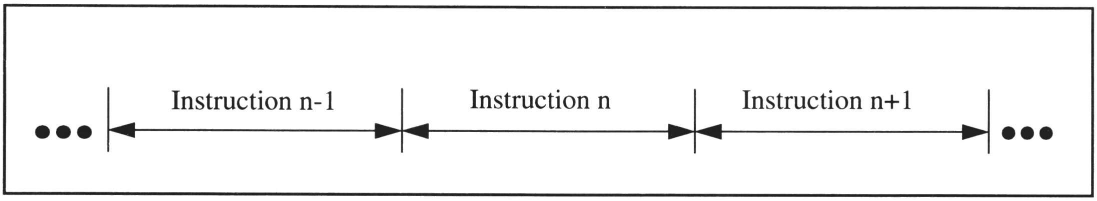

Instruction n-1
Instruction n
Instruction n+1

Figure 4.20 XA Instruction Sequence Overview

A detailed model of this sequence is shown in Figure 4.21: First, at the beginning of the instruction cycle, the state of the TM flag is latched. Next, the instruction is checked to see if it is valid; undefined instructions or disallowed operations (like a write through ES in User Mode) are simply not executed, and there is no chance for a trace to occur. The sequence then checks for instructions illegal in the current context (currently only an IRET while in User Mode is detected here) and services an exception if one is found. If, and only if, none of these special conditions occur, the instruction is actually executed. Just after execution, if the Trace Mode bit had been latched TRUE at the beginning of the instruction cycle, the Trace is serviced. Finally, the cycle checks for a pending interrupt and performs interrupt service if one is found

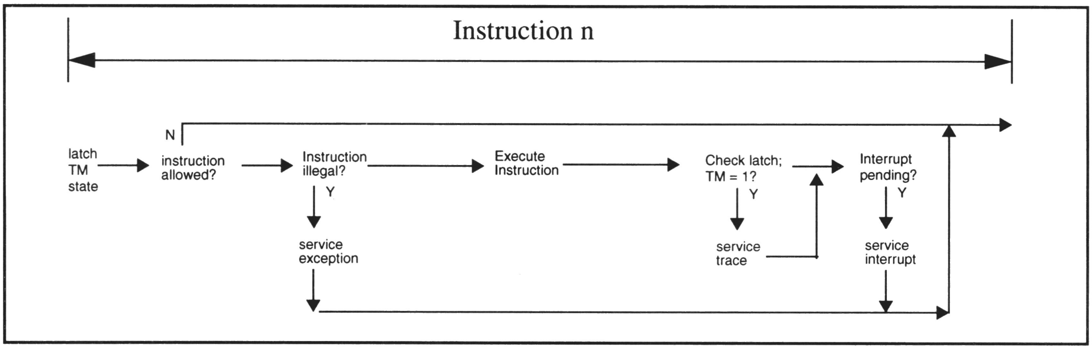

Instruction n
N
latch TM state
instruction allowed?
Instruction illegal?
Y
service exception
Execute Instruction
Check latch; TM = 1?
Y
service trace
Interrupt pending?
Y
service interrupt

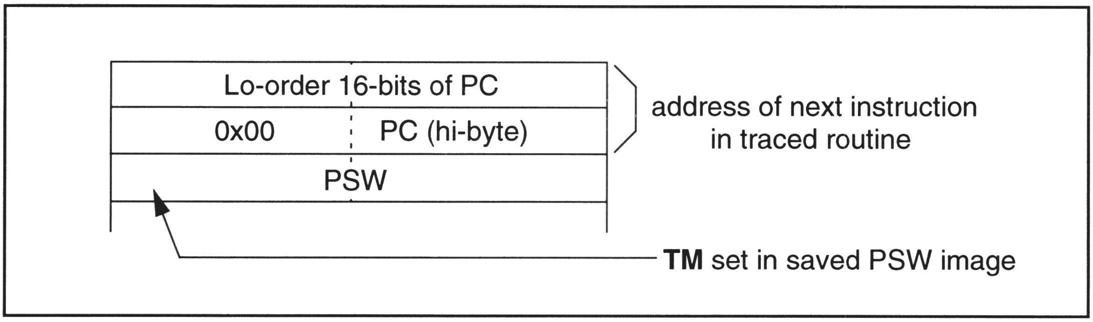
Figure 4.21 Instruction Execution Cycle Detail

Note that an external reset may occur at any point during the cycle illustrated in Figure 4.21. This will abort processing when it occurs.

One consequence of this sequence is that the instruction that sets TM = 1 cannot generate a Trace, since TM is not latched when the instruction is actually executed. Another consequence is that an instruction that generates an exception will never be traced. Finally if an event interrupt occurs during an instruction cycle when the instruction being executed is a TRAP, the TRAP will be executed, then the trace service, and finally the interrupt will be serviced.

4.9.2 Trace Mode Initialization and Deactivation

Since **PSW.TM** is in the protected portion of the PSW (i.e., in PSWH), only code executing in System Mode can initiate or turn off Trace Mode. In practice, this may be done by invoking a trap whose replacement PSW clears this bit, or by executing a RETI instruction with a synthetic Exception/Interrupt stack frame explicitly pushed on the top of the System Stack, as follows:


Lo-order 16-bits of PC
0x00
PC (hi-byte)
PSW
address of next instruction in traced routine
TM set in saved PSW image

Tracing will continue until the PSW bit **TM** is cleared. This may be done by the trace service routine by examining the stack frame at the top of the system stack and clearing the TM bit prior to returning to the currently traced process. A similar method may be used to initiate trace mode. Note that stack frames generated by exception interrupts are always placed on the System stack. It is probably a good idea for the trace service routine to verify that the item in the stack frame is consistent with the traced process before modifying the TM bit.

5 Real-time Multitasking

Multi-tasking as the name suggests, allows tasks, which are pieces of code that do specific duties, to run in an apparently concurrent manner. This means that tasks will seem to all run at the same time, doing many specific jobs simultaneously.

High end applications (like automotive) require instantaneous responses when dealing with high speed events, such as engine management, traction control and adaptive braking system (ABS) and hence there is a trend towards multi-tasking in a wide variety of high performance embedded control applications.

Real-time application programs are often comprised of multiple tasks. Each task manages a specific facet of application program. Building a real-time application from individual tasks allows subdividing a complicated application program into independent and manageable modules. Each task shares the processor with other tasks in the application program according to an assigned priority level.

In real-time multi-tasking, the main concern is the *system overhead.* Switching tasks involve moving lots of data of the terminated and initiated tasks, and extensive book-keeping to be able to restore dormant tasks when required. Thus it is extremely crucial to minimize the system overhead as much as possible. In some cases, some of the tasks may be associated with real-time response, which further complicates the requirements from the system.

The following section analyzes the requirements and the XA suitability to these applications.

5.1 Assist For Multitasking in XA

The XA has numerous provisions to support multi-tasking systems. The architecture provides direct support for the concept of a multi-tasking OS by providing two (System/User) privilege levels for isolation between tasks. High performance, interrupt driven, multi-tasking applications systems requiring protection are feasible with the XA.

The XA architecture offers the following features which will appeal to multi-tasking implementations.

5.1.1 Dual stack approach

The architecture defines a System Stack Pointer (SSP) as well as an User Stack Pointer (USP). The dual stack feature supports fast task switching, and ease the creation of a multi-tasking monitor kernel. The separation of the two offers a reduction is storing and retrieving stack pointers or using a single stack, when switching to the kernel and back to an application. It also serves to speed up interrupt processing in large systems with external data memory. User stacks can be allocated in the slower external memory, while system memory is in internal SRAM, allowing for fast interrupt latency in this environment. The dual stack approach also adds the benefit of a better potential to recover from an ill-behaved task, since the system stack is still intact when an error is sensed.

5.1.2 Register Banks

The XA also supports 4 banks of 8 byte/4 word registers, in addition to 12 shared registers. In some applications, the register banks can be designated statically to tasks, cutting significantly on the overhead for saving and restoring registers on context switching.

5.1.3 Interrupt Latency and Overhead

Interrupt latency is extremely critical in a multitasking environment. For a real-time multitasking environment, a fast interrupt response is crucial for switching between tasks. The XA is designed to provide such fast task switching environment through improved interrupt latency time.

The interrupt service mechanism saves the PC (1 or 2 words, depending on the Page0 mode flag PZ) and the PSW (1 word) on the stack. The interrupt stack normally resides in the internal data memory, so saving 3 words takes only 6 clocks. Prefetching the service routine takes 3 additional clocks. The overhead through the external interrupt controller is around 3 clocks, to allow synchronization and avoid metastability.

When interrupt or an exception/trap occurs, the current instruction in progress always gets executed prior servicing the interrupt. This present an overhead, while increasing the effective interrupt latency, since the event that interrupted the machine cannot be dealt with before the book-keeping is completed. In XA, the longest uninterrupted instruction is the signed 32x16 Divide, which takes 24 clocks.

This puts the worst case interrupt latency at (24+6+3+3 =) 36 clocks (2.25 microsecond at 16 MHz, 1.8 at 20 MHz and 0.83 at 30 MHz). Saving the state of the machine can be done simply by switching the register bank, which takes 3 additional clock.

In the general case, up to 16 registers would be saved on the stack, which takes 32 clocks. The total latency+overhead at start of an interrupt is a maximum of 68 clocks (4.25 microsecond at 16 MHz, 3.4 at 20 MHz and 2.27 at 30 MHz). This allows for extremely fast context switching for multitasking environments.

5.1.4 Protection

The issue is mentioned here simply to clarify what is and what is not supported by the XA architecture. Dual stack pointer and minor privileges to what looks like a supervisor mode do not mean full protection. It is assumed that code in a microcontroller does not require guarding from intentional system break-in by a lower privilege task. A table of the protected features in XA is given below.

Protected Features in the XA

Table: Segment and Stack Register Protection

Mode	Write to DS	Write through DS	Write to ES	Write through ES	Read through DS	Read through ES	Read through SSP	Write to SSP	Write to SSEL bit 7
System	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
User	Disallowed	Allowed	Allowed	Selectable [1]	Allowed	Allowed	Not possible	Not possible	Disallowed

Note 1: The MSB of SSEL (bit 7) selects whether write through ES is allowed in User mode. However, this bit is accessible only in System mode.

Table: PSW bit protection

Mode	Write to SM bit	Write to RS0:1 bits	Write to TM bit	Write to IM0:3 bits
System	Allowed	Allowed	Allowed	Allowed
User	Disallowed	Allowed	Disallowed	Disallowed

Protection Via Data Memory Segmentation

In User/Application mode, each task is protected from all others via the separation of data spaces (unless explicit sharing is planned in advance). If the address spaces of two tasks include no shared data, one task cannot affect the data of another, but it can read any data in the full address space. Code sharing is always safe since code memory may never be written[1]. An application mode program is prohibited from writing the segment registers, thus confining the writable area per an ill-behaved task to its dedicated segment. Most applications, which are not expected to utilize multi-tasking or use external memory, do not require any protection. They will remain after reset in system mode, and could access all system resources.

At any given instant, two segments of memory are immediately accessible to an executing XA program. These are the data segment DS, where the stack and local variables reside, and the extra segment ES, which may be used to read remote data structures. Restricting the addressability of task modules helps gain complete control of system resources for efficient, reliable operation in a multi-tasking environment.

---

[1] True for non-writable code memory only like Eprom, ROM, OTP. This might change for FLASH parts

Protection Via Dual Stack Pointers

The XA provides a two-level user/supervisor protection mechanism. These are the *user* or *application* mode and the *system* or *supervisor* mode. In a multitasking environment, tasks in a supervisor level are protected from tasks in the application level.

The XA has two stack pointers (in the register file) called the System Stack Pointer (SSP) and the User Stack Pointer (USP). In multitasking systems one stack pointer is used for the supervisory system and another for the currently active task. This helps in the protection mechanism by providing isolation of system software from user applications. The two stack pointers also help to improve the performance of interrupts. If the stack for a particular application would exceed the space available in the on-chip RAM, or on-chip RAM is needed for other time critical purposes (since on-chip RAM is accessed more quickly than off-chip memory), the main stack can be put off-chip and the interrupt stack (using the System SP) may be put in on-chip RAM.

These features of the XA place it well above the competition in suitability to multi-tasking applications.

6 Instruction Set and Addressing

This section contains information about the addressing modes and data types used in the XA. The intent is to help the user become familiar with the programming capabilities of the processor.

6.1 Addressing Modes

Addressing modes are ways to form effective addresses of the operands. The XA provides seven *basic* powerful addressing modes for access on word, byte, and bit data, or to specify the target address of a branch instruction. These *basic* addressing modes are uniformly available on a large number of instructions. Table 6-1 includes the basic addressing modes in the XA. An instruction could use a combination of these basic addressing modes, e.g., ADD R0, #020 is a combination of Register and Immediate addressing modes.

All modes (non-register) generate ADDR[15:0]. This address is combined with DS/ES[23:16] for data and PC/CS[23:16] for code to form a 24-bit address[1].

An XA instruction can have zero, one, two, or three operands, whose locations are defined by the addressing mode. A *destination* operand is one that is replaced by a result, or is in some way affected by the instruction. The destination operand is listed first in an addressing mode expression. A *source* operand is a value that is moved or manipulated by the instruction, but is not altered. The source is listed second in an addressing mode expression.

Table 6.1 Basic Addressing Modes

MODE	MNEMONIC	OPERANDS
Register	R	operand(s) in register (in Register file)
Indirect	[R]	Byte/Word whose 16-bit address is in R
Indirect-Offset	[R+off 8/16]	Byte or Word data whose address (16-bit) contained in R, is offset by 8/16-bit signed integer "off 8/16'
Direct	mem_addr	Byte/Word at given memory "mem_addr'
SFR [1]	sfr_addr	Byte/Word at "sfr_addr' address
Immediate	#data 4/5 #data 8/16	Immediate 4/5 and 8/16-bit integer constants "data8/16"
Bit	bit	10-bit address field specifying Register File, Data Memory or SFR bit address space

1. This is a special case of direct addressing mode but separately identified, as SFR space is separate from data memory.

---

[1] Exception is Page 0 mode, where all addresses are 16-bit.

6.2 Description of the Modes

6.2.1 Register Addressing

Instructions using this addressing mode contain a field that addresses the Register File that contains an operand. The Register file is byte[2], word, double-word or bit addressable.

Example: ADD R6, R4

Before: R4 contains 005Ah
R6 contains A5A5h

After: R4 contains 005Ah
R6 contains A5FFh

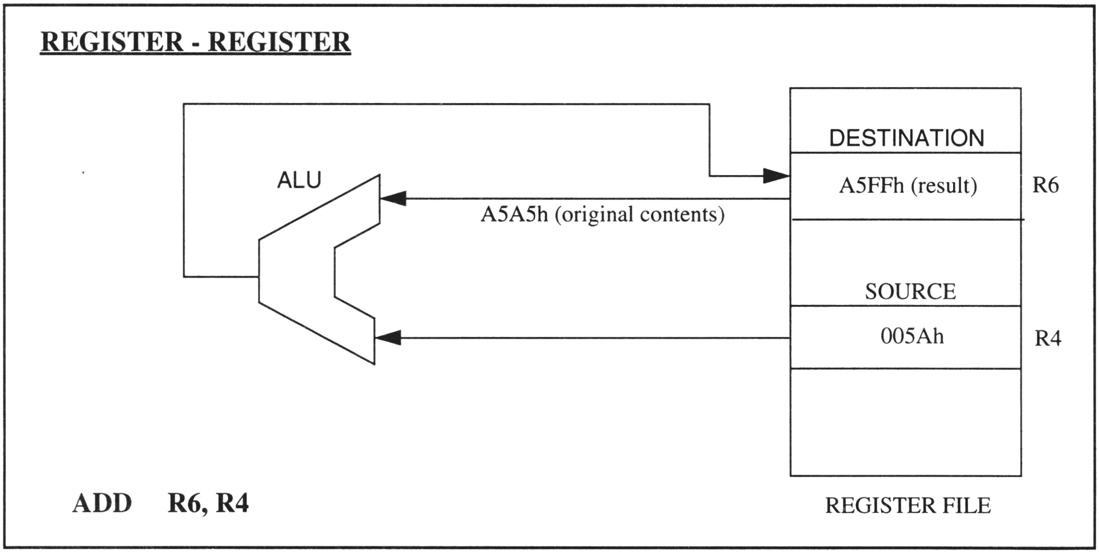

REGISTER - REGISTER
ALU
A5A5h (original contents)
DESTINATION
A5FFh (result)
R6
SOURCE
005Ah
R4
ADD R6, R4
REGISTER FILE

Figure 6.1

---

[2] The unimplemented 8 word registers are not Byte addressable

6.2.2 Indirect Addressing

Instructions using this addressing mode contain a 16-bit address field. This field is contained in 1 out of 8 pointer registers in the Register File (that contain the 16-bit address of the operand in any 64K data segment). For data, the segment is identified by the 8-bit contents of DS or the ES and for code by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL corresponding to the indirect register number. The address of the pointer word for word operands should be even

Example:	ADD R6, [R4]	Before:	R6 contains 1005h
	SSEL.4 = 1		R4 contains A000h
	i.e., the operand is in segment determined		Word at A000h contains A5A5h
	by the contents of ES	After:	R4 contains A000h
	So, if ES = 08, the		R6 contains B5AAh
	operand is in		Word at A000h in segment 8
	segment 8 of data memory.		of data memory contains A5A5h

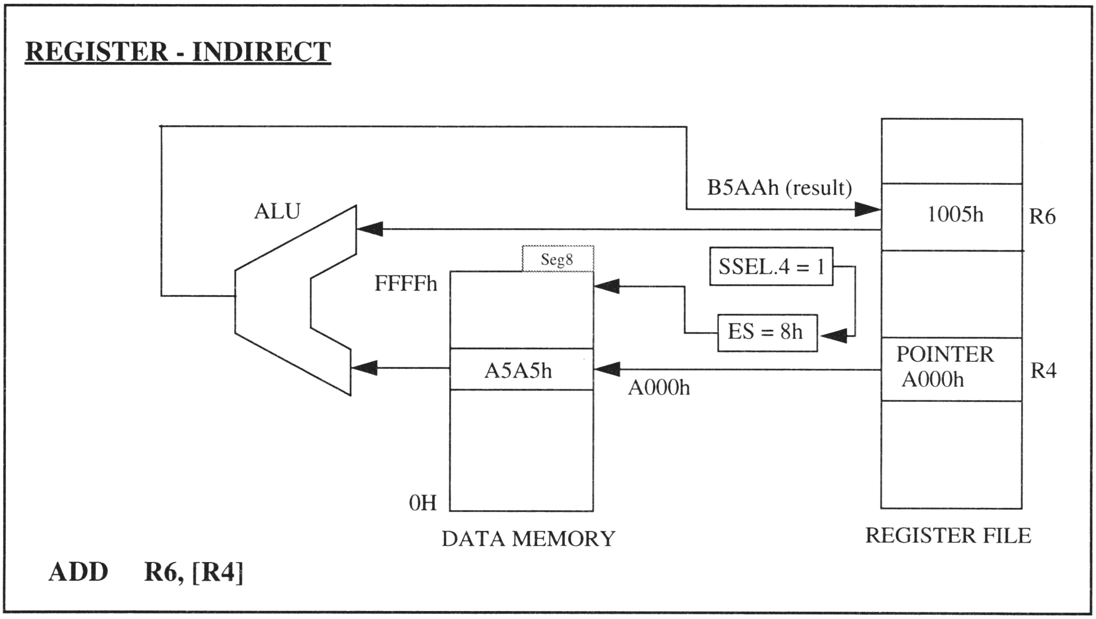

REGISTER - INDIRECT
ALU
B5AAh (result)
1005h
R6
Seg8
FFFFh
SSEL.4 = 1
ES = 8h
A5A5h
POINTER
A000h
R4
A000h
0H
DATA MEMORY
REGISTER FILE
ADD R6, [R4]

Figure 6.2

6.2.3 Indirect-Offset Addressing

This addressing mode is just like the Register-Indirect addressing mode above except that an additional displacement value is added to obtain the final effective address. Instructions using this addressing mode contain a 16-bit address field and an 8 or 16-bit signed displacement field. This field addresses 1 out of 8 pointer registers in the Register File that contains the 16-bit address of the operand in any 64K data segment. The contents of the pointer register are added to the signed displacement to obtain the effective address[3] (which *must* be even) of the operand. For data the segment is identified by the 8-bit contents of DS or the ES and for code, by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL.

Example:	ADD R5, [R3 +30h] SSEL.3 = 1 i.e., the operand is in segment determined	Before:	R3 contains C000h R5 contains 0065h Word at C030h = A540h
	by the contents of ES So, if ES = 04, the operand is in segment 4 of data memory.	After:	R3 contains C000h R5 contains A5A5h Word at C030h = A540h

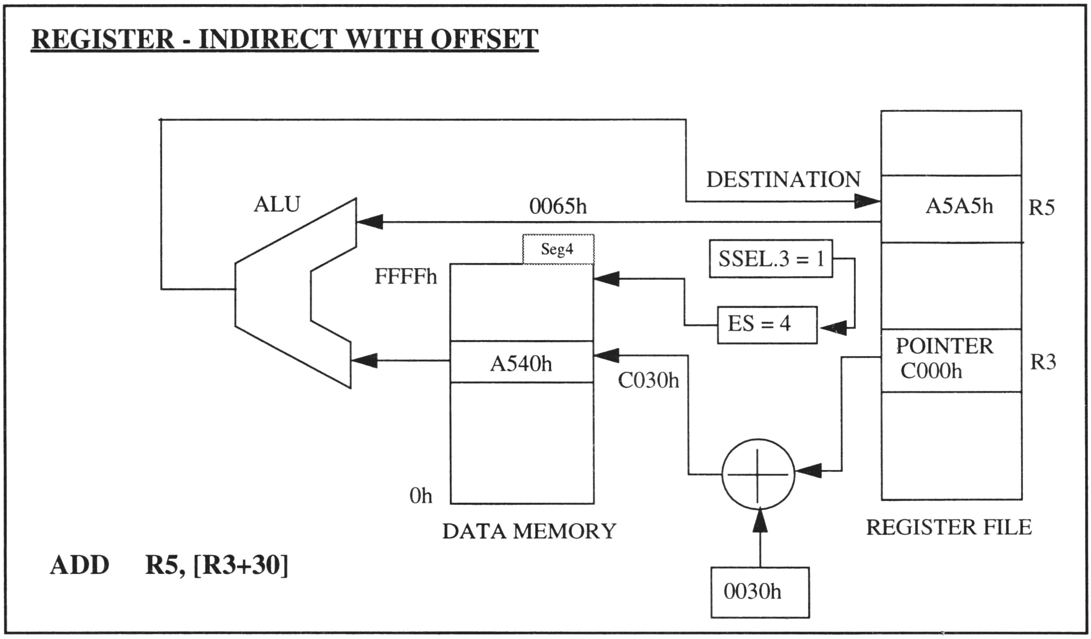

REGISTER - INDIRECT WITH OFFSET
ALU
DESTINATION
0065h
A5A5h
R5
Seg4
SSEL.3 = 1
FFFFh
ES = 4
A540h
POINTER
C000h
R3
C030h
0h
DATA MEMORY
REGISTER FILE
ADD R5, [R3+30]
0030h

Figure 6.3

---

[3] In case of an odd address, the XA forces the operand fetch from the next lower even boundary (address.bit0 = 0)

6.2.4 Direct Addressing

Instructions using this addressing mode contain an 10-bit address field, which contains the actual address of the operand in any 64K data memory segment or sfr space.The direct address data memory space is always the bottom 1K byte (0:3FFh) of any segment. The associated data segment is always identified by the 8-bit contents of DS.

Example:	SUB R0, 200h If DS = 02, the operand is in segment 2 of data memory.	Before:	R0 contains A5FFh 200H contains 5555h
		After:	R0 contains 50AAh 200h contains 5555h

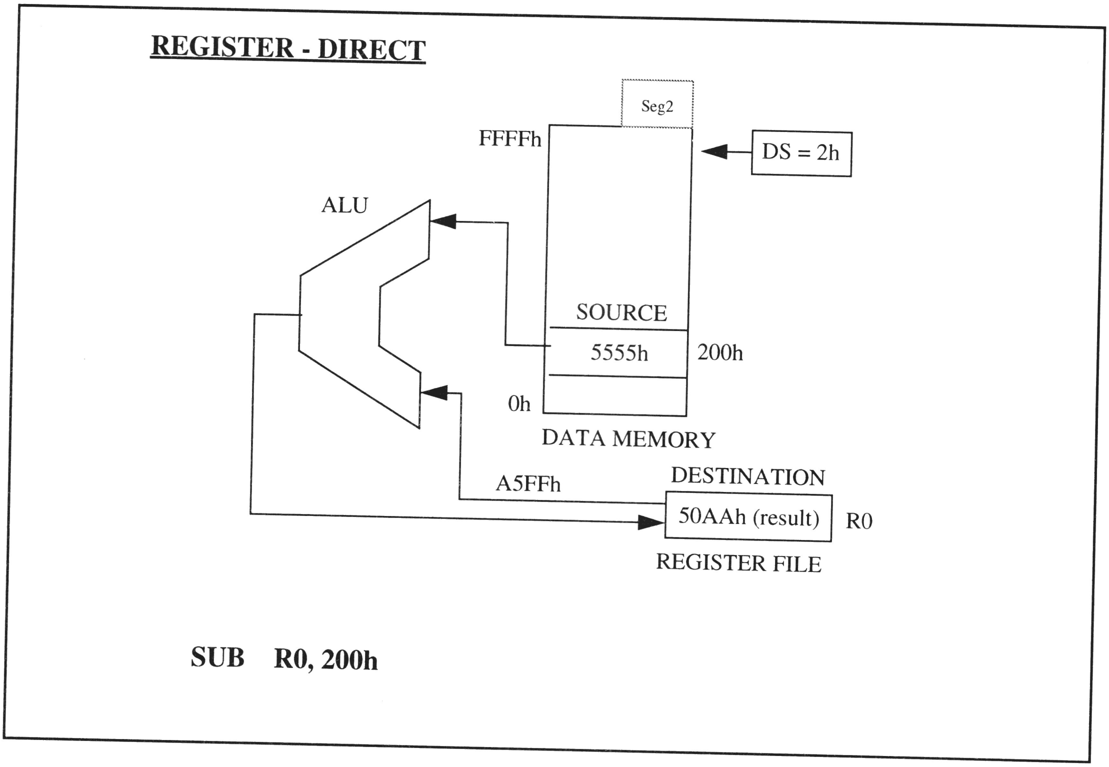

REGISTER - DIRECT
Seg2
FFFFh
DS = 2h
ALU
SOURCE
5555h
200h
0h
DATA MEMORY
A5FFh
DESTINATION
50AAh (result)
R0
REGISTER FILE
SUB R0, 200h

Figure 6.4

6.2.5 SFR Addressing

This is identical to the direct addressing mode described before, except it addresses the 1K SFR space. Although encoded into the same instruction field as the direct addressing described above, this is actually a separate space. Instructions using this addressing mode contain an 10-bit SFR address. The 1K SFR space is always directly addressed (400:7FFh) and is mapped directly above the 1K direct-addressed RAM space.

Example: MOV R0H, 406h[4]

Before: R0H contains 05h
406h contains A5h

After: R0H contains A5h
406h contains A5h

6.2.6 Immediate Addressing

In immediate addressing, the actual operand is given explicitly in the instruction.The immediate operand is either an 4/5, 8 or 16-bit integer which constitutes the source operand. 4-bit short immediate operands used with instructions ADDS and MOVS are sign extended.

Example: ADD R0L,#0B9h

Before: R0 contains 13h
After: R0L contains CCh

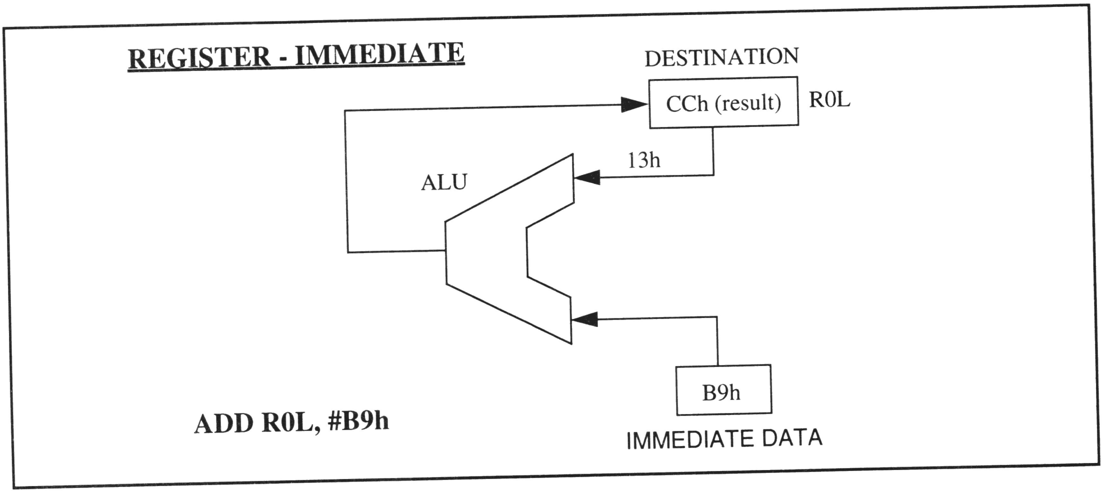


Figure 6.5

---

[4] The syntax always refers to the SFR address starting from the base address of 400H.

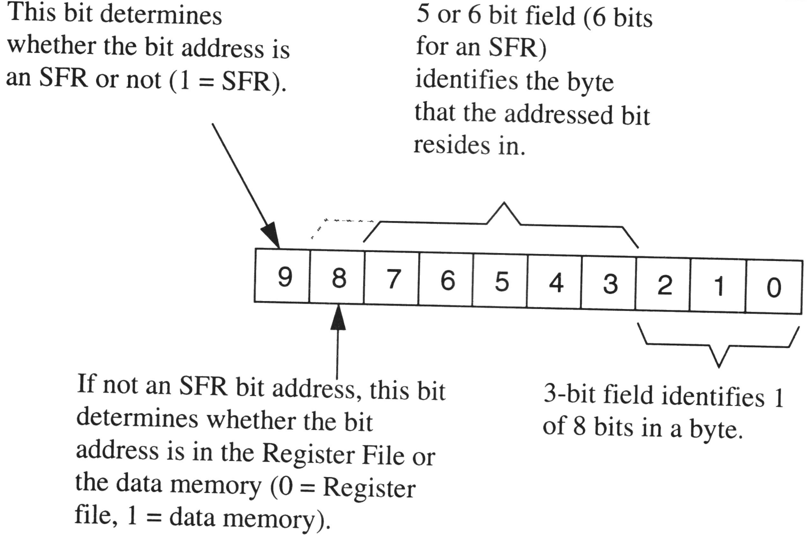
6.2.7 Bit Addressing

Instructions using the bit addressing mode contain a 10-bit field containing the address of the bit operand. The XA supports three bit address spaces, which are encoded into the same format. The spaces are: 256 bits in the register file (the entire active register file); 256 bits in the data memory (byte addresses 20 through 3F hex on the current data segment); and 512 bits in the SFR space (byte addresses 400 through 43F hex).

Bit addresses 0 to FF hex map to the register file, bit addresses 100 to 1FF hex map to data memory, and bit addresses 200 to 3FF map to the SFR space.

A separate bit-addressable space (20-3F hex) in the direct-address data memory, exists for each segment. The current working segment for the direct-address space being always identified by the DS register.

The encoding of the 10-bit field for bit addresses is as follows:


This bit determines whether the bit address is an SFR or not (1 = SFR).
5 or 6 bit field (6 bits for an SFR) identifies the byte that the addressed bit resides in.
9
8
7
6
5
4
3
2
1
0
If not an SFR bit address, this bit determines whether the bit address is in the Register File or the data memory (0 = Register file, 1 = data memory).
3-bit field identifies 1 of 8 bits in a byte.

Bit Address Encoding

Examples:
For a given data segment,
1 001100 010 = Bit 2 of an SFR at address 0Ch (i.e., 40Ch in the map)
0 001100 010 = Bit 2 of Register file at address 0Ch, i.e., R6H
0 101100 010 = Bit 2 of Data memory address 0Ch

Figure 6.6

6.3 Relative Branching and Jumps

Program memory addresses as referenced by Jumps, Calls, and Branch instructions must be word aligned in XA. For instance, a branch instruction may occur at any code address, but it may only branch to an even address. This forced alignment to even address provides three benefits:

- Branch ranges are doubled without providing an extra bit in the instruction and
- Faster execution as XA always fetches first two byte of an instruction simultaneously.
- Allows translated 8051 code to have branches extended over intervening code that will tend to grow when translated and generally increase the chances of a branch target being in that range.

The *rel8* displacement is a 9-bit twos complement integer which is encoded as 8-bits that represents the relative distance in words from the current PC to the destination PC. Similarly, the *rel16* displacement is a 17-bit twos complement integer which is encoded as 16-bits. The value of the PC used in the target address calculation is the address of the instruction following the Branch, Jump or Call instruction.

The 8-bit signed displacement is between -128 to +127. The branch range for rel8 is (sample calculation shown below) is really +254 bytes to -256 bytes for instructions located at an *even* address, and +253 to -257 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

The 16-bit signed displacement is -32,768 to +32,767. The branch range is therefore +65,534 bytes to -65,536 bytes for instructions located at an *even* address, and +65,533 to -65,537 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

Sample calculation for rel8 range:

Assuming word aligned branch target, forward range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = +127; So, rel8 << 1 is +254

If Current PC = ODD, then
Range = 254 - 1 = +253 as PC is forced to an even location, else
If current PC = EVEN, then
Range = +254

Similarly, reverse range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = -128; So, rel8 << 1 is -256

If Current PC = ODD, then
Range = -257
Else if current PC = EVEN, then
Range = -256

6.4 Data Types in XA

The XA uses the following types of data:

- Bits
- 4/5-bit signed integers
- 8-bit (byte) signed and unsigned integers
- 8-bit, two digit BCD numbers
- 16-bit (word) signed and unsigned integers
- 10-bit address for bit-addressing in data memory and SFR space
- 24-bit effective address comprising of 16-bit address and 8-bit segment select. See addressing modes for more information.

A byte consists of 8-bits. A word is a 16-bit value consisting of two contiguous bytes. A double word consists of two 16-bit words packed in two contiguous words in memory.

Negative integers are represented in twos complement form. 4-bit signed integers (sign extended to byte/word) are used as immediate operands in MOVS and ADDS instructions.

Binary coded decimal numbers are packed, 2 digits per byte. BCD operations use byte operands.

6.5 Instruction Set Overview

The XA uses a powerful and efficient instruction set, offering several different types of addressing modes. A versatile set of "branch" and "jump" instructions are available for controlling program flow based on register or memory contents. Special emphasis has been placed on the instruction support of structured high-level languages and real-time multi-tasking operating systems.

This section discusses the set of instructions provided in the XA microcontroller, and also shows how to use them. It includes descriptions of the instruction format and the operands used by the instructions. After a summary of the instructions by category, the section provides a detailed description of the operation of each instruction, in alphabetical order.

Five summary tables are provided that describes the available instructions. The first table is a summary of instructions available in the XA along with their common usage. The second and third table are tables of addressing modes and operands, and the instruction type they pertain to. A fourth table that lists the summary of status flags update by different instructions. A fifth table lists the available instructions with their different addressing modes and briefly describes what each instruction does along with the number of bytes, and number of cycles required for each instruction.

The formats have been chosen to optimize the length and execution speed of those instructions that would be used the most often in critical code. Only the first and sometimes the second byte of an instruction are used for operation encoding. The length of the instruction and the first execution cycle activity are determined from the first byte. Instruction bytes following the first two bytes (if any) are always immediate operands, such as addresses, relative displacements, offsets, bit addresses, and immediate data.

Glossary of mnemonics, notations used

General:

offset8	An 8-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
offset16	A 16-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
direct	An 11-bit immediate address contained in the instruction.
#data4	4 bits of immediate data contained in the instruction. (range +7 to -8 for signed immediate data and 0-15 for shifts)
#data5	5 bits of immediate data contained in the instruction. (0-31 for shifts)
#data8	8 bits of immediate data contained in the instruction. (+127 to -128)
#data16	16 bits of immediate data contained in the instruction. (+32,767 to -32,768)
bit	The 10-bit address of an addressable bit.
rel8	An 8-bit relative displacement for branches. (+254 to -256)
rel16	An 16-bit relative displacement for branches.(+65,534 to -65,536)
addr16	A 16-bit absolute branch address within a 64K code page.
addr24	A 24-bit absolute branch address, able to access the entire XA address space.
SP	The current Stack Pointer (User or System) depending on the operation mode.
USP	The User Stack Pointer.
SSP	The System Stack Pointer
C	Carry flag from the PSW.
AC	Auxiliary Carry flag from the PSW.
V	Overflow flag from the PSW.
N	Negative flag from the PSW.
Z	Zero flag from the PSW.

Operation encoding fields:

DS	Data Size. This field encodes whether the operation is byte, word or double-word.
IND	This field flags indirect operation in some instructions.
H/L	This field selects whether PUSH and POP Rlist use the upper or lower half of the available registers.
dddd	Destination register field, specifies one of 16 registers in the register file.
ddd	Destination register field for indirect references, specifies one of 8 pointer registers in the register file.
ssss	Source register field, specifies one of 16 registers in the register file.
sss	Source register field for indirect references, specifies one of 8 pointer registers in the register file.

Mnemonic text:

Rs	Source register.
Rd	Destination register.
[ ]	In the instruction mnemonic, indicates an indirect reference (e.g.: [R4] refers to the memory address pointed to by the contents of register 4).
[R+]	Used to indicate an automatic increment of the pointer register in some indirect addressing modes.
[WS:R]	Indicates that the pointer register (R) is extended to a 24-bit pointer by the selected segment register (either DS or ES for all instructions except MOVC, which uses either $PC_{23\text{-}16}$ or CS).

Pseudocode:

( )	Used to indicate "contents of" in the instruction operation pseudocode (e.g.: (R4) refers to the contents of register 4).
<---	Pseudocode assignment operator. Occasionally used as <--> to indicate assignment in both directions (interchange of data).
((SP))	Data memory contents at the location pointed to by the current stack pointer. In system mode, the current SP is the SSP, and the segment used is always segment 0. In user mode, the current SP is the USP, and the segment used is the Data Segment (DS). This segment apply to the uses of the SP, not just PUSH and POP. In a few cases, "((SSP))" or "((USP))" indicate that a specific SP is used, regardless of the operating mode.

Rn.x	Indicates bit x of register n.
Rn.x-y	Indicates a range of bits from bit x to bit y of register n.

Note: all indirect addressing is accomplished using the contents of the data segment register as the upper 8 address bits unless otherwise specified. Example: [ES:Rs] indicates that the extra segment register generates the upper 8 bits of the address in this case.

Cycle time:

PZ - In Page 0
nt - Not Taken
t - Taken
tbd - To be determined

Syntax For Operand size:

.w = For word operands
.b = byte operands
.d = double-word operands

Default operand size is dependant on the operands used e.g MOV R0,R1 is always word-size whereas MOV R0L, R0H is always byte etc. For INDIRECT_IMMEDIATE, DIRECT_IMMEDIATE, DIRECT_DIRECT, etc., user must specify operand size.

Others

0x = prefix for Hex values
[] = For indirect addressing
[[]] = For Double-indirect addressing
dest = destination
src = source

Table 6.2 Instruction Set in XA

Mnemonic	Usage
MOV, MOVC, MOVS, MOVX, LEA, XCH, PUSH, POP, PUSHU, POPU	Data Movement
ADD, ADDS, ADDC, SUB, SUBB	Add and Subtract
MULU.b, MULU.w, MUL.w DIVU.b, DIVU.w, DIVU.d, DIV.w, DIV.d	Multiply and Divide
RR, RRC, RL, RLC, LSR, ASR, ASL, NORM	Shifts and Rotates
CLR, SETB, MOV, ANL, ORL	Bit Operations
JB, JBC, JNB, JNZ, JZ, DJNZ, CJNE,	Conditional Jumps/Calls
BOV, BNV, BPL, BCC, BCS, BEQ, BNE, BG, BGE, BGT, BL, BLE, BLT, BMI	Conditional Branches
AND, OR, XOR	Boolean Functions
JMP, FJMP, CALL, FCALL, BR	Unconditional Jumps/Calls/Branches
RET, RETI	Return from subroutines, interrupts
SEXT, NEG, CPL, DA	Sign Extend, Negate, Complement, Decimal Adjust
BKPT, TRAP#, RESET	Exceptions
NOP	No Operation

Table 6.3 shows a summary of the basic addressing modes available for data transfer and calculation related instructions.

Table 6.3 Instruction Addressing Modes

Modes/ Operands	MOVX	MOV	CMP	ADD ADDC	SUB SUBB	AND OR XOR	ADDS MOVS	MUL DIV	Shift	XCH	bytes
R, R		•	•	•	•	•		•	•	•	2
R, [R]	•	•	•	•	•	•				•	2
[R], R	•	•	•	•	•	•					2
R, [R+off8]		•	•	•	•	•					3
[R+off8], R		•	•	•	•	•					3
R, [R+off16]		•	•	•	•	•					4
[R+off16], R		•	•	•	•	•					4
R, [R+]		•	•	•	•	•					2
[R+], R		•	•	•	•	•					2
[R+], [R+]		•									2
dir, R		•	•	•	•	•					3
R, dir		•	•	•	•	•				•	3
dir, [R]		•									3
[R], dir		•									3
R, #data		•	•	•	•	•	•	•	•		2*/3/4
[R], #data		•	•	•	•	•	•				2*/3/4
[R+], #data		•	•	•	•	•	•				2*/3/4
[R+off8], #data		•	•	•	•	•	•				3*/4/5
[R+off16], #data		•	•	•	•	•	•				4*/5/6
dir, #data		•	•	•	•	•	•				3*/4/5
dir, dir		•									4
R, USP		•									2
R, [R+]	•								2		
[R+],R	•								2		
A, [A+DPTR]	•								2		
A, [A+PC]	•								2		
direct		•							3		
Rlist		•							2		

Table 6.3 Instruction Addressing Modes

Modes/ Operands	MOVX	MOV	CMP	ADD ADDC	SUB SUBB	AND OR XOR	ADDS MOVS	MUL DIV	Shift	XCH	bytes
R			•						2		
addr24				•					4		
[R]				•					2		
[A+DPTR]				JMP					2		
R, rel					•				3		
direct, rel					•				4		
R, direct, rel						•			4		
R, #data, rel						•			4/5		
[R], #data, rel						•			4/5		
bit							•		3		
bit, C; C, bit							•		3		
C, /bit							•		3		
rel				•				Cond. Branch	2		

Notes:

- Shift class includes rotates, shifts, and normalize.
- USP = User stack pointer.

* : ADDS and MOVS uses a short immediate field (4 bits).

** instructions with no operands include: BKPT, NOP, RESET, RET, RETI.

Table 6.4 summarizes the status flag updates for the various XA instruction types.

Table 6.4 Status Flag Updates

Instruction Type	Flags Updated				
	C	AC	V	N	Z
ADD, ADDC, CMP, SUB, SUBB	X	X	X	X	X
ADDS, MOVS	-	-	-	X	X
AND, OR, XOR	-	-	-	X	X
ASR, LSR	*	-	-	X	X
branches, all bit operations, NOP	-	-	-	-	-
Calls, Jumps, and Returns	-	-	-	-	-
CJNE	X	-	-	X	X
CPL	-	-	-	X	X
DA	*	-	-	X	X
DIV, MUL	*	-	*	X	X
DJNZ	-	-	-	X	X
LEA	-	-	-	-	-
MOV, MOVC, MOVX	-	-	-	X	X
NEG	-	-	X	X	X
NORM	-	-	-	X	X
PUSH, POP	-	-	-	-	-
RESET	*	*	*	*	*
RL, RR	-	-	-	X	X
RLC, RRC	*	-	-	X	X
SEXT	-	-	-	-	-
TRAP, BKPT	-	-	-	-	-
XCH	-	-	-	-	-
ASL	*	-	X	X	X

Notes:
-: flag not updated.
X: flag updated according to the standard definition.
*: flag update is non-standard, refer to the individual instruction description.
Note: Explicit writes to PSW flags takes precedence over flag updates.

Instruction Set Summary

Table 6.5 lists the entire XA instruction set by instruction type. This can be used as a quick reference to find specific instructions that may be looked up in the detailed alphabetical description section.

Table 6.5

Mnemonic		Description	Bytes	Clocks
Arithmetic Operations				
ADD	Rd, Rs	Add registers direct	2	3
ADD	Rd, [Rs]	Add register-indirect to register	2	4
ADD	[Rd], Rs	Add register to register-indirect	2	4
ADD	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register	3	6
ADD	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset	3	6
ADD	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register	4	6
ADD	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset	4	6
ADD	Rd, [Rs+]	Add register-indirect with auto increment to register	2	5
ADD	[Rd+], Rs	Add register-indirect with auto increment to register	2	5
ADD	direct, Rs	Add register to memory	3	4
ADD	Rd, direct	Add memory to register	3	4
ADD	Rd, #data8	Add 8-bit immediate data to register	3	3
ADD	Rd, #data16	Add 16-bit immediate data to register	4	3
ADD	[Rd], #data8	Add 8-bit immediate data to register-indirect	3	4
ADD	[Rd], #data16	Add 16-bit immediate data to register-indirect	4	4
ADD	[Rd+], #data8	Add 8-bit immediate data to register-indirect with auto-increment	3	5
ADD	[Rd+], #data16	Add 16-bit immediate data to register-indirect with auto-increment	4	5
ADD	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset	4	6
ADD	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADD	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset	5	6
ADD	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset	6	6
ADD	direct, #data8	Add 8-bit immediate data to memory	4	4
ADD	direct, #data16	Add 16-bit immediate data to memory	5	4
ADDC	Rd, Rs	Add registers direct with carry	2	3
ADDC	Rd, [Rs]	Add register-indirect to register with carry	2	4
ADDC	[Rd], Rs	Add register to register-indirect with carry	2	4
ADDC	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register with carry	3	6
ADDC	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset with carry	3	6
ADDC	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register with carry	4	6
ADDC	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset with carry	4	6
ADDC	Rd, [Rs+]	Add register-indirect with auto increment to register with carry	2	5
ADDC	[Rd+], Rs	Add register-indirect with auto increment to register with carry	2	5
ADDC	direct, Rs	Add register to memory with carry	3	4
ADDC	Rd, direct	Add memory to register with carry	3	4
ADDC	Rd, #data8	Add 8-bit immediate data to register with carry	3	3
ADDC	Rd, #data16	Add 16-bit immediate data to register with carry	4	3
ADDC	[Rd], #data8	Add 16-bit immediate data to register-indirect with carry	3	4
ADDC	[Rd], #data16	Add 16-bit immediate data to register-indirect with carry	4	4
ADDC	[Rd+], #data8	Add 8-bit immediate data to register-indirect and auto-increment with carry	3	5
ADDC	[Rd+], #data16	Add 16-bit immediate data to register-indirect and auto-increment with carry	4	5
ADDC	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset and carry	4	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADDC	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset and carry	5	6
ADDC	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset and carry	5	6
ADDC	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset and carry	6	6
ADDC	direct, #data8	Add 8-bit immediate data to memory with carry	4	4
ADDC	direct, #data16	Add 16-bit immediate data to memory with carry	5	4
ADDS	Rd, #data4	Add 4-bit signed immediate data to register	2	3
ADDS	[Rd], #data4	Add 4-bit signed immediate data to register-indirect	2	4
ADDS	[Rd+], #data4	Add 4-bit signed immediate data to register-indirect with auto-increment	2	5
ADDS	[Rd+offset8], #data4	Add register-indirect with 8-bit offset to 4-bit signed immediate data	3	6
ADDS	[Rd+offset16], #data4	Add register-indirect with 16-bit offset to 4-bit signed immediate data	4	6
ADDS	direct, #data4	Add 4-bit signed immediate data to memory	3	4
ASL	Rd, Rs	Logical left shift destination register by the value in the source register	2	See Note1
ASL	Rd, #data4	Logical left shift register by the 4-bit immediate value	2	See Note1
ASR	Rd, Rs	Arithmetic shift right destination register by the count in the source	2	See Note1
ASR	Rd, #data4	Arithmetic shift right register by the 4-bit immediate count	2	See Note1
CMP	Rd, Rs	Compare destination and source registers	2	3
CMP	[Rd], Rs	Compare register with register-indirect	2	4
CMP	Rd, [Rs]	Compare register-indirect with register	2	4
CMP	[Rd+offset8], Rs	Compare register with register-indirect with 8-bit offset	3	6
CMP	[Rd+offset16], Rs	Compare register with register-indirect with 16-bit offset	4	6
CMP	Rd, [Rs+offset8]	Compare register-indirect with 8-bit offset with register	3	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
CMP	Rd,[Rs+offset16]	Compare register-indirect with 16-bit offset with register	4	6
CMP	Rd, [Rs+]	Compare auto-increment register-indirect with register	2	5
CMP	[Rd+], Rs	Compare register with auto-increment register-indirect	2	5
CMP	direct, Rs	Compare register with memory	3	4
CMP	Rd, direct	Compare memory with register	3	4
CMP	Rd, #data8	Compare 8-bit immediate data to register	3	3
CMP	Rd, #data16	Compare 16-bit immediate data to register	4	3
CMP	[Rd], #data8	Compare 8 -bit immediate data to register-indirect	3	4
CMP	[Rd], #data16	Compare 16-bit immediate data to register-indirect	4	4
CMP	[Rd+], #data8	Compare 8-bit immediate data to register-indirect with auto-increment	3	5
CMP	[Rd+], #data16	Compare 16-bit immediate data to register-indirect with auto-increment	4	5
CMP	[Rd+offset8], #data8	Compare 8-bit immediate data to register-indirect with 8-bit offset	4	6
CMP	[Rd+offset8], #data16	Compare 16-bit immediate data to register-indirect with 8-bit offset	5	6
CMP	[Rd+offset16], #data8	Compare 8-bit immediate data to register-indirect with 16-bit offset	5	6
CMP	[Rd+offset16], #data16	Compare 16-bit immediate data to register-indirect with 16-bit offset	6	6
CMP	direct, #data8	Compare 8-bit immediate data to memory	4	4
CMP	direct, #data16	Compare 16-bit immediate data to memory	5	4
DA	Rd	Decimal Adjust byte register	2	4
DIV.w	Rd, Rs	16x8 signed register divide	2	14
DIV.w	Rd, #data8	16x8 signed divide register with immediate word	3	12
DIV.d	Rd, Rs	32x16 signed double register divide	2	24
DIV.d	Rd, #data16	32x16 signed double register divide with immediate word	4	24
DIVU.b	Rd, Rs	8x8 unsigned register divide	2	12

Table 6.5

Mnemonic		Description	Bytes	Clocks
DIVU.b	Rd, #data8	8X8 unsigned register divide with immediate byte	3	12
DIVU.w	Rd, Rs	16X8 unsigned register divide	2	12
DIVU.w	Rd, #data8	16X8 unsigned register divide with immediate byte	3	12
DIVU.d	Rd, Rs	32X16 unsigned double register divide	2	22
DIVU.d	Rd, #data16	32X16 unsigned double register divide with immediate word	4	22
LEA	Rd, Rs+offset8	Load 16-bit effective address with 8-bit offset to register	3	3
LEA	Rd, Rs+offset16	Load 16-bit effective address with 16-bit offset to register	4	3
MUL.w	Rd, Rs	16X16 signed multiply of register contents	2	12
MUL.w	Rd, #data16	16X16 signed multiply 16-bit immediate data with register	4	12
MULU.b	Rd, Rs	8X8 unsigned multiply of register contents	2	12
MULU.b	Rd, #data8	8X8 unsigned multiply of 8-bit immediate data with register	3	12
MULU.w	Rd, Rs	16X16 unsigned register multiply	2	12
MULU.w	Rd, #data16	16X16 unsigned multiply 16-bit immediate data with register	4	12
NEG	Rd	Negate (twos complement) register	2	3
SEXT	Rd	Sign extend last operation to register	2	3
SUB	Rd, Rs	Subtract registers direct	2	3
SUB	Rd, [Rs]	Subtract register-indirect to register	2	4
SUB	[Rd], Rs	Subtract register to register-indirect	2	4
SUB	Rd, [Rs+offset8]	Subtract register-indirect with 8-bit offset to register	3	6
SUB	[Rd+offset8], Rs	Subtract register to register-indirect with 8-bit offset	3	6
SUB	Rd, [Rs+offset16]	Subtract register-indirect with 16-bit offset to register	4	6
SUB	[Rd+offset16], Rs	Subtract register to register-indirect with 16-bit offset	4	6
SUB	Rd, [Rs+]	Subtract register-indirect with auto increment to register	2	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUB	[Rd+], Rs	Subtract register-indirect with auto increment to register	2	5
SUB	direct, Rs	Subtract register to memory	3	4
SUB	Rd, direct	Subtract memory to register	3	4
SUB	Rd, #data8	Subtract 8-bit immediate data to register	3	3
SUB	Rd, #data16	Subtract 16-bit immediate data to register	4	3
SUB	[Rd], #data8	Subtract 8-bit immediate data to register-indirect	3	4
SUB	[Rd], #data16	Subtract 16-bit immediate data to register-indirect	4	4
SUB	[Rd+], #data8	Subtract 8-bit immediate data to register-indirect with auto-increment	3	5
SUB	[Rd+], #data16	Subtract 16-bit immediate data to register-indirect with auto-increment	4	5
SUB	[Rd+offset8], #data8	Subtract 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUB	[Rd+offset8], #data16	Subtract 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUB	[Rd+offset16], #data8	Subtract 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUB	[Rd+offset16], #data16	Subtract 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUB	direct, #data8	Subtract 8-bit immediate data to memory	4	4
SUB	direct, #data16	Subtract 16-bit immediate data to memory	5	4
SUBB	Rd, Rs	Subtract with borrow registers direct	2	3
SUBB	Rd, [Rs]	Subtract with borrow register-indirect to register	2	4
SUBB	[Rd], Rs	Subtract with borrow register to register-indirect	2	4
SUBB	Rd, [Rs+offset8]	Subtract with borrow register-indirect with 8-bit offset to register	3	6
SUBB	[Rd+offset8], Rs	Subtract with borrow register to register-indirect with 8-bit offset	3	6
SUBB	Rd, [Rs+offset16]	Subtract with borrow register-indirect with 16-bit offset to register	4	6
SUBB	[Rd+offset16], Rs	Subtract with borrow register to register-indirect with 16-bit offset	4	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUBB	Rd, [Rs+]	Subtract with borrow register-indirect with auto increment to register	2	5
SUBB	[Rd+], Rs	Subtract with borrow register-indirect with auto increment to register	2	5
SUBB	direct, Rs	Subtract with borrow register to memory	3	4
SUBB	Rd, direct	Subtract with borrow memory to register	3	4
SUBB	Rd, #data8	Subtract with borrow 8-bit immediate data to register	3	3
SUBB	Rd, #data16	Subtract with borrow 16-bit immediate data to register	4	3
SUBB	[Rd], #data8	Subtract with borrow 8-bit immediate data to register-indirect	3	4
SUBB	[Rd], #data16	Subtract with borrow 16-bit immediate data to register-indirect	4	4
SUBB	[Rd+], #data8	Subtract with borrow 8-bit immediate data to register-indirect with auto-increment	3	5
SUBB	[Rd+], #data16	Subtract with borrow 16-bit immediate data to register-indirect with auto-increment	4	5
SUBB	[Rd+offset8], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUBB	[Rd+offset8], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUBB	[Rd+offset16], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUBB	[Rd+offset16], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUBB	direct, #data8	Subtract with borrow 8-bit immediate data to memory	4	4
SUBB	direct, #data16	Subtract with borrow 16-bit immediate data to memory	5	4
Logical Operations				
AND	Rd, Rs	Logical AND registers direct	2	3
AND	Rd, [Rs]	Logical AND register-indirect to register	2	4
AND	[Rd], Rs	Logical AND register to register-indirect	2	4
AND	Rd, [Rs+offset8]	Logical AND register-indirect with 8-bit offset to register	3	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
AND	[Rd+offset8], Rs	Logical AND register to register-indirect with 8-bit offset	3	6
AND	Rd, [Rs+offset16]	Logical AND register-indirect with 16-bit offset to register	4	6
AND	[Rd+offset16], Rs	Logical AND register to register-indirect with 16-bit offset	4	6
AND	Rd, [Rs+]	Logical AND register-indirect with auto increment to register	2	5
AND	[Rd+], Rs	Logical AND register-indirect with auto increment to register	2	5
AND	direct, Rs	Logical AND register to memory	3	4
AND	Rd, direct	Logical AND memory to register	3	4
AND	Rd, #data8	Logical AND 8-bit immediate data to register	3	3
AND	Rd, #data16	Logical AND 16-bit immediate data to register	4	3
AND	[Rd], #data8	Logical AND 8-bit immediate data to register-indirect	3	4
AND	[Rd], #data16	Logical AND16-bit immediate data to register-indirect	4	4
AND	[Rd+], #data8	Logical AND 8-bit immediate data to register-indirect and auto-increment	3	5
AND	[Rd+], #data16	Logical AND16-bit immediate data to register-indirect and auto-increment	4	5
AND	[Rd+offset8], #data8	Logical AND8-bit immediate data to register-indirect with 8-bit offset	4	6
AND	[Rd+offset8], #data16	Logical AND16-bit immediate data to register-indirect with 8-bit offset	5	6
AND	[Rd+offset16], #data8	Logical AND8-bit immediate data to register-indirect with 16-bit offset	5	6
AND	[Rd+offset16], #data16	Logical AND16-bit immediate data to register-indirect with 16-bit offset	6	6
AND	direct, #data8	Logical AND 8-bit immediate data to memory	4	4
AND	direct, #data16	Logical AND16-bit immediate data to memory	5	4
CPL	Rd	Complement (ones complement) register	2	3
LSR	Rd, Rs	Logical right shift destination register by the value in the source register	2	See Note 1

Table 6.5

Mnemonic		Description	Bytes	Clocks
LSR	Rd, #data4	Logical right shift register by the 4-bit immediate value	2	See Note 1
NORM	Rd, Rs	Logical shift left destination register by the value in the source register until MSB set	2	See Note 1
OR	Rd, Rs	Logical OR registers	2	3
OR	Rd, [Rs]	Logical OR register-indirect to register	2	4
OR	[Rd], Rs	Logical OR register to register-indirect	2	4
OR	Rd, [Rs+offset8]	Logical OR register-indirect with 8-bit offset to register	3	6
OR	[Rd+offset8], Rs	Logical OR register to register-indirect with 8-bit offset	3	6
OR	Rd, [Rs+offset16]	Logical OR register-indirect with 16-bit offset to register	4	6
OR	[Rd+offset16], Rs	Logical OR register to register-indirect with 16-bit offset	4	6
OR	Rd, [Rs+]	Logical OR register-indirect with auto increment to register	2	5
OR	[Rd+], Rs	Logical OR register-indirect with auto increment to register	2	5
OR	direct, Rs	Logical OR register to memory	3	4
OR	Rd, direct	Logical OR memory to register	3	4
OR	Rd, #data8	Logical OR 8-bit immediate data to register	3	3
OR	Rd, #data16	Logical OR 16-bit immediate data to register	4	3
OR	[Rd], #data8	Logical OR 8-bit immediate data to register-indirect	3	4
OR	[Rd], #data16	Logical OR 16-bit immediate data to register-indirect	4	4
OR	[Rd+], #data8	Logical OR 8-bit immediate data to register-indirect with auto-increment	3	5
OR	[Rd+], #data16	Logical OR 16-bit immediate data to register-indirect with auto-increment	4	5
OR	[Rd+offset8], #data8	Logical OR 8-bit immediate data to register-indirect with 8-bit offset	4	6
OR	[Rd+offset8], #data16	Logical OR 16-bit immediate data to register-indirect with 8-bit offset	5	6
OR	[Rd+offset16], #data8	Logical OR 8-bit immediate data to register-indirect with 16-bit offset	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
OR	[Rd+offset16], #data16	Logical OR 16-bit immediate data to register-indirect with 16-bit offset	6	6
OR	direct, #data8	Logical OR 8-bit immediate data to memory	4	4
OR	direct, #data16	Logical OR16-bit immediate data to memory	5	4
RL	Rd, #data4	Rotate left register by the 4-bit immediate value	2	See Note 1
RLC	Rd, #data4	Rotate left register though carry by the 4-bit immediate value	2	See Note 1
RR	Rd, #data4	Rotate right register by the 4-bit immediate value	2	See Note 1
RRC	Rd, #data4	Rotate right register though carry by the 4-bit immediate value	2	See Note 1
XOR	Rd, Rs	Logical XOR registers	2	3
XOR	Rd, [Rs]	Logical XOR register-indirect to register	2	4
XOR	[Rd], Rs	Logical XOR register to register-indirect	2	4
XOR	Rd, [Rs+offset8]	Logical XOR register-indirect with 8-bit offset to register	3	6
XOR	[Rd+offset8], Rs	Logical XOR register to register-indirect with 8-bit offset	3	6
XOR	Rd, [Rs+offset16]	Logical XOR register-indirect with 16-bit offset to register	4	6
XOR	[Rd+offset16], Rs	Logical XOR register to register-indirect with 16-bit offset	4	6
XOR	Rd, [Rs+]	Logical XOR register-indirect with auto increment to register	2	5
XOR	[Rd+], Rs	Logical XOR register-indirect with auto increment to register	2	5
XOR	direct, Rs	Logical XOR register to memory	3	4
XOR	Rd, direct	Logical XOR memory to register	3	4
XOR	Rd, #data8	Logical XOR 8-bit immediate data to register	3	3
XOR	Rd, #data16	Logical XOR 16-bit immediate data to register	4	3
XOR	[Rd], #data8	Logical XOR 8-bit immediate data to register-indirect	3	4
XOR	[Rd], #data16	Logical XOR 16-bit immediate data to register-indirect	4	4

Table 6.5

Mnemonic		Description	Bytes	Clocks
XOR	[Rd+], #data8	Logical XOR 8-bit immediate data to register-indirect with auto-increment	3	5
XOR	[Rd+], #data16	Logical XOR 16-bit immediate data to register-indirect with auto-increment	4	5
XOR	[Rd+offset8], #data8	Logical XOR 8-bit immediate data to register-indirect with 8-bit offset	4	6
XOR	[Rd+offset8], #data16	Logical XOR 16-bit immediate data to register-indirect with 8-bit offset	5	6
XOR	[Rd+offset16], #data8	Logical XOR 8-bit immediate data to register-indirect with 16-bit offset	5	6
XOR	[Rd+offset16], #data16	Logical XOR 16-bit immediate data to register-indirect with 16-bit offset	6	6
XOR	direct, #data8	Logical XOR 8-bit immediate data to memory	4	4
XOR	direct, #data16	Logical XOR16-bit immediate data to memory	5	4
Data transfer				
MOV	Rd, Rs	Move register to register	2	3
MOV	Rd, [Rs]	Move register-indirect to register	2	3
MOV	[Rd], Rs	Move register to register-indirect	2	3
MOV	Rd, [Rs+offset8]	Move register-indirect with 8-bit offset to register	3	5
MOV	[Rd+offset8], Rs	Move register to register-indirect with 8-bit offset	3	5
MOV	Rd, [Rs+offset16]	Move register-indirect with 16-bit offset to register	4	5
MOV	[Rd+offset16], Rs	Move register to register-indirect with 16-bit offset	4	5
MOV	Rd, [Rs+]	Move register-indirect with auto increment to register	2	4
MOV	[Rd+], Rs	Move register-indirect with auto increment to register	2	4
MOV	direct, Rs	Move register to memory	3	4
MOV	Rd, direct	Move memory to register	3	4
MOV	[Rd+], [Rs+]	Move register-indirect to register-indirect, both pointers auto-incremented	2	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOV	direct, [Rs]	Move register-indirect to memory	3	4
MOV	[Rd], direct	Move memory to register-indirect	3	4
MOV	Rd, #data8	Move 8-bit immediate data to register	3	3
MOV	Rd, #data16	Move 16-bit immediate data to register	4	3
MOV	[Rd], #data8	Move 16-bit immediate data to register-indirect	3	3
MOV	[Rd], #data16	Move 16-bit immediate data to register-indirect	4	3
MOV	[Rd+], #data8	Move 8-bit immediate data to register-indirect with auto-increment	3	4
MOV	[Rd+], #data16	Move 16-bit immediate data to register-indirect with auto-increment	4	4
MOV	[Rd+offset8], #data8	Move 8-bit immediate data to register-indirect with 8-bit offset	4	5
MOV	[Rd+offset8], #data16	Move 16-bit immediate data to register-indirect with 8-bit offset	5	5
MOV	[Rd+offset16], #data8	Move 8-bit immediate data to register-indirect with 16-bit offset	5	5
MOV	[Rd+offset16], #data16	Move 16-bit immediate data to register-indirect with 16-bit offset	6	5
MOV	direct, #data8	Move 8-bit immediate data to memory	4	3
MOV	direct, #data16	Move 16-bit immediate data to memory	5	3
MOV	direct, direct	Move memory to memory	4	4
MOV	Rd, USP	Move User Stack Pointer to register (system mode only)	2	3
MOV	USP, Rs	Move register to User Stack Pointer (system mode only)	2	3
MOVC	Rd, [Rs+]	Move data from WS:Rs address of code memory to register with auto-increment	2	4
MOVC	A, [A+DPTR]	Move data from code memory to the accumulator indirect with DPTR	2	6
MOVC	A, [A+PC]	Move data from code memory to the accumulator indirect with PC	2	6
MOVS	Rd, #data4	Move 4-bit sign-extended immediate data to register	2	3
MOVS	[Rd], #data4	Add 4-bit sign-extended immediate data to register-indirect	2	4

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOVS	[Rd+], #data4	Add 4-bit sign-extended immediate data to register-indirect with auto-increment	2	4
MOVS	[Rd+offset8], #data4	Add register-indirect with 8-bit offset to 4-bit sign-extended immediate data	3	5
MOVS	[Rd+offset16], #data4	Add register-indirect with 16-bit offset to 4-bit sign-extended immediate data	4	5
MOVS	direct, #data4	Add 4-bit sign-extended immediate data to memory	3	3
MOVX	Rd, [Rs]	Move external data from memory to register	2	6
MOVX	[Rd], Rs	Move external data from register to memory	2	6
PUSH	direct	Push the memory content (byte/word) onto the current stack	3	5
PUSHU	direct	Push the memory content (byte/word) onto the user stack	3	5
PUSH	Rlist	Push multiple registers (byte/word) onto the current stack	2	See Note 2
PUSHU	Rlist	Push multiple registers (byte/word)from the user stack	2	See Note 2
POP	direct	Pop the memory content (byte/word) from the current stack	3	5
POPU	direct	Pop the memory content (byte/word) from the user stack	3	5
POP	Rlist	Pop multiple registers (byte/word) from the current stack	2	See Note 3
POPU	Rlist	Pop multiple registers (byte/word) from the user stack	2	See Note 3
XCH	Rd, Rs	Exchange contents of two registers	2	5
XCH	Rd, [Rs]	Exchange contents of a register-indirect address with a register	2	6
XCH	Rd, direct	Exchange contents of memory with a register	3	6
Program Branching				
BCC	rel8	Branch if the carry flag is clear	2	6t/3nt
BCS	rel8	Branch if the carry flag is set	2	6t/3nt
BEQ	rel8	Branch if the zero flag is set	2	6t/3nt
BNE	rel8	Branch if the zero flag is not set	2	6t/3nt

Table 6.5

Mnemonic		Description	Bytes	Clocks
BG	rel8	Branch if greater than (unsigned)	2	6t/3nt
BGE	rel8	Branch if greater than or equal to (signed)	2	6t/3nt
BGT	rel8	Branch if greater than (signed)	2	6t/3nt
BL	rel8	Branch if less than or equal to (unsigned)	2	6t/3nt
BLE	rel8	Branch if less than or equal to (signed)	2	6t/3nt
BLT	rel8	Branch if less than (signed)	2	6t/3nt
BMI	rel8	Branch if the negative flag is set	2	6t/3nt
BPL	rel8	Branch if the negative flag is clear	2	6t/3nt
BNV	rel8	Branch if overflow flag is clear	2	6t/3nt
BOV	rel8	Branch if overflow flag is set	2	6t/3nt
BR	rel8	Short unconditional branch	2	3
CALL	[Rs]	Subroutine call indirect with a register	2	8/5(PZ)
CALL	rel16	Relative call (+/- 64K)	3	7/4(PZ)
CJNE	Rd,direct,rel8	Compare direct byte to register and jump if not equal	4	10t/7nt
CJNE	Rd,#data8,rel8	Compare immediate byte to register and jump if not equal	4	9t/6nt
CJNE	Rd,#data16,rel8	Compare immediate word to register and jump if not equal	5	9t/6nt
CJNE	[Rd],#data8,rel8	Compare immediate word to register-indirect and jump if not equal	4	10t/7nt
CJNE	[Rd],#data16,rel8	Compare immediate word to register-indirect and jump if not equal	5	10t/7nt
DJNZ	Rd,rel8	Decrement register and jump if not zero	3	8t/5nt
DJNZ	direct,rel8	Decrement memory and jump if not zero	4	9t/6nt
FCALL	addr24	Far call (anywhere in the 24-bit address space)	4	9/5(PZ)
FJMP	addr24	Far jump (anywhere in the 24-bit address space)	4	6
JB	bit,rel8	Jump if bit set	4	7t/4nt
JBC	bit,rel8	Jump if bit set and then clear the bit	4	7t/4nt
JMP	rel16	Long unconditional branch	3	6
JMP	[Rs]	Jump indirect to the address in the register (64K)	2	3

Table 6.5

Mnemonic		Description	Bytes	Clocks
JMP	[A+DPTR]	Jump indirect relative to the DPTR	2	5
JMP	[[Rs+]]	Jump double-indirect to the address (pointer to a pointer)	2	8
JNB	bit,rel8	Jump if bit not set	4	7t/4nt
JNZ	rel8	Jump if accumulator not equal zero	2	7t/4nt
JZ	rel8	Jump if accumulator equals zero	2	7t/4nt
NOP		No operation	1	3
RET		Return from subroutine	2	8/6(PZ)
RETI		Return from interrupt	2	10/ 8(PZ)
Bit Manipulation				
ANL	C, bit	Logical AND bit to carry	3	4
ANL	C, /bit	Logical AND complement of a bit to carry	3	4
CLR	bit	Clear bit	3	4
MOV	C, bit	Move bit to the carry flag	3	4
MOV	bit, C	Move carry to bit	3	4
ORL	C, bit	Logical OR a bit to carry	3	4
ORL	C, /bit	Logical OR complement of a bit to carry	3	4
SETB	bit	Sets the bit specified	3	4
Exception / Trap				
BKPT		Cause the breakpoint trap to be executed.	1	23/ 19(PZ)
RESET		Causes a hardware Reset, identical to an external Reset	2	8
TRAP	#data4	Causes 1 of 16 hardware traps to be executed	2	23/ 19(PZ)

Note 1: For 8 and 16 bit shifts, it is 4+1 per additional two bits. For 32-bit shifts, it is 6+1 per additional two bits.
Note 2: 3 clocks + 3 clocks/register.
Note 3: 4 clocks +2 clocks/register.

ADD Integer Addition

Syntax: ADD dest, source

Operation: dest <- src + dest

Description: Performs a twos complement binary addition of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

Note: If used with write to PSWL, takes precedence to flag updates

Sizes: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

ADD Rd, Rs

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rd) + (Rs)
Encoding:

0	0	0	0	DS	0	0	1

d	d	d	d	s	s	s	s

ADD Rd, [Rs]

Bytes: 2
Cycles: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:

0	0	0	0	DS	0	1	0

d	d	d	d	0	s	s	s

ADD [Rd], Rs

Bytes: 2
Cycles: 4
Operation: (Rd) <-- (Rd) + (Rs)
Encoding:

0	0	0	0	DS	0	1	0

s	s	s	s	1	d	d	d

ADD Rd, [Rs+offset8]

Bytes: 3
Cycles: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:

0	0	0	0	DS	1	0	0		d	d	d	d	0	s	s	s

byte 3: offset8

ADD [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:

0	0	0	0	DS	1	0	0		s	s	s	s	1	d	d	d

byte 3: offset8

ADD Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:

0	0	0	0	DS	1	0	1		d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADD [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:

0	0	0	0	DS	1	0	1		s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADD Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	0	DS	0	1	1

d	d	d	d	0	s	s	s

ADD [Rd+], Rs

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	0	DS	0	1	1

s	s	s	s	1	d	d	d

ADD direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:

0	0	0	0	DS	1	1	0

s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

ADD Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:

0	0	0	0	DS	1	1	0

d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

ADD Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:

1	0	0	1	0	0	0	1		d	d	d	d	0	0	0	0

byte 3: #data8

ADD Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:

1	0	0	1	1	0	0	1		d	d	d	d	0	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:

1	0	0	1	0	0	1	0		0	d	d	d	0	0	0	0

byte 3: #data8

ADD [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:

1	0	0	1	1	0	1	0		0	d	d	d	0	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:

1	0	0	1	0	0	1	1	0	d	d	d	0	0	0	0

byte 3: #data8

ADD [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1	0	d	d	d	0	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:

1	0	0	1	0	1	0	0	0	d	d	d	0	0	0	0

byte 3: offset8
byte 4: #data8

ADD [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:

1	0	0	1	1	1	0	0	0	d	d	d	0	0	0	0

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADD [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	0	0	0	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

ADD [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	0	0	0	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

ADD direct, #data8
Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct) + #data8
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	0	0	0	0

byte 3: lower 8 bits of direct
byte 4: #data8

ADD direct, #data16

Bytes: 5
Cycles: 4
Operation: (direct) <-- (direct) + #data16
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	0	0	0

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADDC Integer addition with Carry

Syntax: ADD dest, source

Operation: dest <- dest + src + C

Description: Performs a twos complement binary addition of the source operand and the previously generated carry bit with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is one (C=1), the result is greater than the sum of the operands; if it is zero (C=0), the result is the exact sum.

This form of addition is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then ADDC is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

ADDC Rd, Rs

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rd) + (Rs) + (C)
Encoding:

0	0	0	1	DS	0	0	1

d	d	d	d	s	s	s	s

ADDC Rd, [Rs]

Bytes: 2
Cycles: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
Encoding:

0	0	0	1	DS	0	1	0

d	d	d	d	0	s	s	s

ADDC [Rd], Rs

Bytes: 2
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
Encoding:

0	0	0	1	DS	0	1	0

s	s	s	s	1	d	d	d

ADDC Rd, [Rs+offset8]

Bytes: 3
Cycles: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8) + (C)
Encoding:

0	0	0	1	DS	1	0	0

d	d	d	d	0	s	s	s

byte 3: offset8

ADDC [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs) + (C)
Encoding:

0	0	0	1	DS	1	0	0

s	s	s	s	1	d	d	d

byte 3: offset8

ADDC Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16) + (C)
Encoding:

0	0	0	1	DS	1	0	1

d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDC [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs) + (C)
Encoding:

0	0	0	1	DS	1	0	1		s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDC Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	1	DS	0	1	1		d	d	d	d	0	s	s	s

ADDC [Rd+], Rs

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	1	DS	0	1	1		s	s	s	s	1	d	d	d

ADDC direct, Rs

Bytes: 3
Cycles: 6
Operation: (direct) <-- (direct) + (Rs) + (C)
Encoding:

0	0	0	1	DS	1	1	0		s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

ADDC Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (Rd) + (direct) + (C)
Encoding:

0	0	0	1	DS	1	1	0

d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

ADDC Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- (Rd) + #data8 + (C)
Encoding:

1	0	0	1	0	0	0	1

d	d	d	d	0	0	0	1

byte 3: #data8

ADDC Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rd) + #data16 + (C)
Encoding:

1	0	0	1	1	0	0	1

d	d	d	d	0	0	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd], #data8

Bytes: 3
Cycles : 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
Encoding:

1	0	0	1	0	0	1	0

0	d	d	d	0	0	0	1

byte 3: #data8

ADDC [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)

Encoding:

1	0	0	1	1	0	1	0		0	d	d	d	0	0	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
(Rd) <-- (Rd) + 1

Encoding:

1	0	0	1	0	0	1	1		0	d	d	d	0	0	0	1

byte 3: #data8

ADDC [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1		0	d	d	d	0	0	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8 + (C)
Encoding:

1	0	0	1	0	1	0	0		0	d	d	d	0	0	0	1

byte 3: offset8
byte 4: #data8

ADDC [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16 + (C)
Encoding:

1	0	0	1	1	1	0	0

0	d	d	d	0	0	0	1

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8 + (C)
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	0	0	0	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

ADDC [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16 + (C)
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	0	0	0	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

ADDC direct, #data8

Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct) + #data8 + (C)
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	0	0	0	1

byte 3: lower 8 bits of direct
byte 4: #data8

ADDC direct, #data16

Bytes:	5
Cycles:	4
Operation:	(direct) <-- (direct) + #data16 + (C)
Encoding:	

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	0	0	1

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADDS Add Short

Syntax: ADDS dest, #value

Operation: dest <- dest + #data4

Description: Four bits of signed immediate data are added to the destination. The immediate data is sign-extended to the proper size, then added to the variable specified by the destination operand, which may be either a byte or a word. The immediate data range is +7 to -8. This instruction is used primarily to increment or decrement pointers and counters.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

(Note: the C and AC flags must not be updated by ADDS since this instruction is used to replace the 80C51 INC and DEC instructions, which do not update the flags.)

ADDS Rd, #data4

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rd) + #data4
Encoding:

1	0	1	0	DS	0	0	1

d	d	d	d	#data4

ADDS [Rd], #data4

Bytes: 2
Cycles: 4
Operation:((WS:Rd)) <-- ((WS:Rd)) + #data4
Encoding:

1	0	1	0	DS	0	1	0

0	d	d	d	#data4

ADDS [Rd+], #data4

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

1	0	1	0	DS	0	1	1

0	d	d	d	#data4

ADDS [Rd+offset8], #data4

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data4
Encoding:

1	0	1	0	DS	1	0	0

0	d	d	d	#data4

byte 3: offset8

ADDS [Rd+offset16], #data4

Bytes: 4
Cycles: 6
Operation:((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data4
Encoding:

1	0	1	0	DS	1	0	1

0	d	d	d	#data4

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDS direct, #data4

Bytes: 3
Cycles: 4
Operation:(direct) <-- (direct) + #data4

Encoding:

1	0	1	0	DS	1	1	0

0	direct: 3 bits	#data4

byte 3: lower 8 bits of direct

AND Logical AND

Syntax: AND dest, src

Operation: dest <- dest AND src

Description: Bitwise logical AND the contents of the source to the destination. The byte or word specified by the source operand is logically ANDed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

AND Rd, Rs

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rd) • (Rs)
Encoding:

0	1	0	1	DS	0	0	1

d	d	d	d	s	s	s	s

AND Rd, [Rs]

Bytes: 2
Cycles: 4
Operation: (Rd) <-- (Rd) • ((WS:Rs))
Encoding:

0	1	0	1	DS	0	1	0

d	d	d	d	0	s	s	s

AND [Rd], Rs

Bytes: 2
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
Encoding:

0	1	0	1	DS	0	1	0

s	s	s	s	1	d	d	d

AND Rd, [Rs+offset8]

Bytes: 3
Cycles: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset8)
Encoding:

0	1	0	1	DS	1	0	0

d	d	d	d	0	s	s	s

byte 3: offset8

AND [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • (Rs)
Encoding:

0	1	0	1	DS	1	0	0

s	s	s	s	1	d	d	d

byte 3: offset8

AND Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset16)
Encoding:

0	1	0	1	DS	1	0	1

d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • (Rs)
Encoding:

0	1	0	1	DS	1	0	1

s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) <-- (Rd) • ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:

0	1	0	1	DS	0	1	1		d	d	d	d	0	s	s	s

AND [Rd+], Rs

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	1	0	1	DS	0	1	1		s	s	s	s	1	d	d	d

AND direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) <-- (direct) • (Rs)
Encoding:

0	1	0	1	DS	1	1	0		s	s	s	s	1	direct: 3 bits		

byte 3: lower 8 bits of direct

AND Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (Rd) • (direct)
Encoding:

0	1	0	1	DS	1	1	0		d	d	d	d	0	direct: 3 bits		

byte 3: lower 8 bits of direct

AND Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- (Rd) • #data8
Encoding:

1	0	0	1	0	0	0	1		d	d	d	d	0	1	0	1

byte 3: #data8

AND Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rd) • #data16
Encoding:

1	0	0	1	1	0	0	1		d	d	d	d	0	1	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
Encoding:

1	0	0	1	0	0	1	0		0	d	d	d	0	1	0	1

byte 3: #data8

AND [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
Encoding:

1	0	0	1	1	0	1	0		0	d	d	d	0	1	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
(Rd) <-- (Rd) + 1
Encoding:

1	0	0	1	0	0	1	1

0	d	d	d	0	1	0	1

byte 3: #data8

AND [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1

0	d	d	d	0	1	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data8
Encoding:

1	0	0	1	0	1	0	0

0	d	d	d	0	1	0	1

byte 3: offset8
byte 4: #data8

AND [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data16
Encoding:

1	0	0	1	1	1	0	0

0	d	d	d	0	1	0	1

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

AND [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data8
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	0	1	0	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

AND [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data16
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	0	1	0	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

AND direct, #data8

Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct) • #data8
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	0	1	0	1

byte 3: lower 8 bits of direct
byte 4: #data8

AND direct, #data16

Bytes: 5
Cycles: 4
Operation: (direct) <-- (direct) • #data16
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	1	0	1

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ANL Logical AND a bit to the Carry flag

Syntax: ANL C, bit

Operation: C <- C (AND) Bit

Description: Read the specified bit and logically AND it to the Carry flag.

Size: Bit

Flags Updated: none

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	0	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

ANL Logical AND the complement of a bit to the Carry flag

Syntax: ANL C, /bit

Operation: Carry <- C (AND) $\overline{\text{bit}}$

Description: Read the specified bit, complement it, and logically AND it to the Carry flag.

Size: Bit

Flags Updated: C

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	0	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

Syntax: ASL dest, count

Operation:

```
Do While (count not equal to 0)
(C) <- (dest.msb)
(dest.bit n+1) <- (dest.bit n)
count = count-1
if sign change during shift,
(V) <- 1
End While
```

Description:
If the count operand is greater than 0, the destination operand is logically shifted left by the number of bits specified by the count operand. The Low-order bits shifted in are zero-filled and the high-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.

The count operand could be:
- An immediate value (#data4 or #data5)
- A Register (Only 5-bits are used to implement up to 32 bit shifts)

The count is a positive value which may be from 1 to 31 and the destination operand is a signed integer (twos complement form).The destination operand (data size) may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.The count operand is not affected by the operation.

Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the shifting is truncated to 5 bits i.e 32 else for immediate shift count, shifting is continued until count is 0.

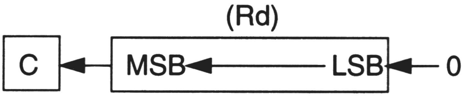
Size: Byte, word, and double word

Flags Updated: C, V, N, Z

Note: the V flag is set if the sign changes anytime during the shift operation and remains set till the end of the shift operation i.e the V flag does not get cleared even if the sign reverts to its original state because of continued shifts.

ASL Rd, Rs

Operation:



Bytes: 2

Cycles: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding:

1	1	0	0	DS1	DS0	0	1

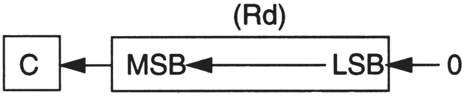
d	d	d	d	s	s	s	s

ASL Rd, #data4
Rd,#data5

Bytes: 2

Cycles: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Operation:



Encoding: (for byte and word data sizes)

1	1	0	0	DS1	DS0	1	0

d	d	d	d	#data4

(for double word data size)

1	1	0	1	1	1	0	1

d	d	d	#data5

Note: DS/DS = 00 : byte operation; DS/DS = 10 : word operation; DS/DS = 11 : double word operation.

Note: The ASL clears the V flag if the condition to set it does not occur.

Syntax: ASR dest, count

Operation:

```
Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
dest.msb <- Sign bit
count = count-1
End While
```

Description:
If the count operand is greater than 0, the destination operand is logically shifted right by the number of bits specified by the count operand. The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed. To preserve the sign of the original operand, the MSBs of the result are sign-extended with the sign bit.

The count operand could be:
- An immediate value (#data4/5)
- A Register (Only 5-bits are used to implement up to 32 bit shifts)

The count operand could be an immediate value or a register. The count is a positive value which may be from 0 to 31 and the destination operand is a signed integer. The count operand is not affected by the operation. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.

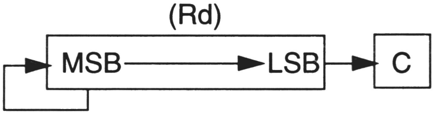
Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the shifting is truncated to 5 bits i.e 32 else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte, Word, Double Word

Flags Updated: C, N, Z

ASR Rd, Rs

Bytes: 2
Cycles: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift
Operation:

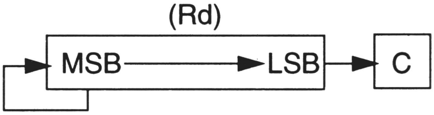


Encoding:

1	1	0	0	DS1	DS0	1	0	d	d	d	d	s	s	s	s

ASR Rd, #data4
Rd,#data5

Operation:



Bytes: 2
Cycles: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding: (for byte and word data sizes)

1	1	0	1	DS1	DS0	1	0	d	d	d	d	#data4

(for double word data size)

1	1	0	1	DS1	DS0	1	0	d	d	d	#data5

Note: DS1/DS0 = 00: byte operation; DS1/DS0 = 10: word operation; DS1/DS0 = 11: double word operation.

BCC Branch if carry clear

Syntax: BCC rel8

Operation:

```
(PC) <-- (PC) + 2
if (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) did not generate a carry (the carry flag contains a 0). If Carry is clear, the program execution branches at the location of the PC, plus the specified displacement, rel8. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6 (t) / 3 (nt)

Encoding:

1	1	1	1	0	0	0	0	rel8

BCS Branch if carry set

Syntax: BCS rel8

Operation:

```
(PC) <-- (PC) + 2
if (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) generated a carry (the carry flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	0	0	0	1		rel8

BEQ Branch if zero

Syntax: BEQ rel8

Operation:

```
(PC) <-- (PC) + 2
if (Z) = 1 then
    (PC) <-- (PC + rel8*2)
    (PC.0) <-- 0
```

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a result of zero (the Z flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	0	0	1	1		rel8

BG Branch if greater than (unsigned)

Syntax: BG rel8

Operation:

```
(PC) <-- (PC) + 2
if (Z) OR (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes:	2
Cycles:	6t/3nt

Encoding:

1	1	1	1	1	0	0	0	rel8

BGE Branch if greater than or equal to (signed)

Syntax: BGE rel8

Operation:

```
(PC) <-- (PC) + 2
if (N) XOR (V) = 0 then
    (PC) <-- (PC + rel8*2)
    (PC.0) <-- 0
```

Description: The branch is taken if the last compare instruction had a destination value that was greater than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	1	0	1	0		rel8

BGT **Branch if greater than (signed)**

Syntax: BGT rel8

Operation: (PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	1	1	0	0	rel8

BKPT Breakpoint

Syntax: BKPT

Operation:
(PC) <-- (PC) + 1
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (bkpt vector)
(PC.15-0) <-- code memory (bkpt vector)
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes a breakpoint trap. The breakpoint trap acts like an immediate interrupt, using a vector to call a specific piece of code that will be executed in system mode. This instruction is intended for use in emulator systems to provide a simple method of implementing hardware breakpoints.

For a breakpoint to work properly under all conditions, it must have an instruction length no greater than the smallest other instruction on the processor, in this case the one byte NOP. This requirement exists because a breakpoint may be inserted in place of a NOP that is followed by another instruction that is branched to or otherwise executed without going through the breakpoint. If the breakpoint instruction were longer than the NOP, it would corrupt the next instruction in sequence if that instruction were executed.

The opcode for the breakpoint instruction is specifically assigned to be all ones (FFh). This is so that un-programmed EPROM code memory will contain breakpoints. Similarly, the NOP instruction is assigned to opcode 00 so that both "blank" code states map to innocuous instructions.

Size: None

Flags Updated: none[5]

Bytes: 1
Cycles: 23/19 (PZ)

Encoding:

1	1	1	1	1	1	1	1

---

[5] All flags are affected during the PSW load from the vector table. It is possible that these flags are restored by the debugger, but does not have to be the case.

BL Branch if less than or equal to (unsigned)

Syntax: BL rel8

Operation:

```
(PC) <-- (PC) + 2
if (Z) OR (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	1	0	0	1	rel8

BLE Branch if less than or equal (signed)

Syntax: BLE rel8

Operation:

```
(PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	1	1	0	1		rel8

BLT Branch if less than (signed)

Syntax: BLT rel8

Operation:

```
(PC) <-- (PC) + 2
if (N) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last compare instruction had a destination value that was less than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	1	0	1	1	rel8

BMI Branch if negative

Syntax: BMI rel8

Operation:

```
(PC) <-- (PC) + 2
if (N) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is less than 0 (the N flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	0	1	1	1		rel8

BNE **Branch if not equal**

Syntax: BNE rel8

Operation:

```
(PC) <-- (PC) + 2
if (Z) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a non-zero result (the Z flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	0	0	1	0	rel8

BNV Branch if no overflow

Syntax: BNV rel8

Operation:

```
(PC) <-- (PC) + 2
if (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0
```

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) did not generate an overflow (The V flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	0	1	0	0	rel8

BOV Branch if overflow flag

Syntax: BOV rel8

Operation:
(PC) <-- (PC) + 2
if (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) generated an overflow (the V flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	0	1	0	1	rel8

BPL Branch if positive

Syntax: BPL rel8

Operation: (PC) <-- (PC) + 2
if (N) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is greater than 0 (the N flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	0	1	1	0		rel8

BR Unconditional Branch

Syntax: BR rel8

Operation:
(PC) <-- (PC) + 2
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: None

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	1	1	1	1	0	rel8

CALL Call Subroutine Relative

Syntax: CALL rel16

Operation:
(PC) <-- (PC) + 3
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +65,534 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory. The 24-bit return address is saved on the stack.

Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Note: Refer to section 6.3 for details of branch range

Size: None

Flags Updated: none

Bytes: 3
Cycles: 7/4(PZ)

Encoding:

1	1	0	0	0	1	0	1

byte 2: upper 8 bits of rel16
byte 3: lower 8 bits of rel16

CALL Call Subroutine Indirect

Syntax: CALL [Rs]

Operation:
(PC) <-- (PC) + 2
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC.15-1) <-- (Rs.15-1)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address contained in the operand register, anywhere within the 64K page following the CALL instruction.The return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned, as CALL or branch will force PC.bit0 to 0.

Note:
(1) Since the PC always points to the instruction following the CALL instruction and if that happens to be on a different page, then the called routine should be located in that page (64K)

(2) if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 2
Cycles: 8/5(PZ)

Encoding:

1	1	0	0	0	1	1	0

0	0	0	0	0	s	s	s

CJNE Compare and jump if not equal

Syntax: CJNE dest, src, rel8

Operation: (PC) <-- (PC) + # of instruction bytes

(dest) - (direct) (result not stored)

if (Z) = 0 then

(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: The byte or word specified by the source operand is compared to the variable specified by the destination operand and the status flags are updated. Jump to the specified address if the values are not equal. The source and destination data are not affected by the operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Byte-Byte, Word-Word

Flags Updated: C, N, Z

(Note: this particular type of compare must not update the V or AC flags to duplicate the 80C51 function.)

CJNE Rd, direct, rel8

Bytes: 4

Cycles: 10t/7nt

Encoding:

1	1	1	0	DS	0	1	0

d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

byte 4: rel8

CJNE Rd, #data8, rel8

Bytes: 4
Cycles: 9t/6nt

Encoding:

1	1	1	0	0	0	1	1

d	d	d	d	0	0	0	0

byte 3: rel8
byte 4: data#8

CJNE Rd, #data16, rel8

Bytes: 5
Cycles: 9t/6nt

Encoding:

1	1	1	0	1	0	1	1

d	d	d	d	0	0	0	0

byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CJNE [Rd], #data8, rel8

Bytes: 4
Cycles: 10t/7nt
Encoding:

1	1	1	0	0	0	1	1

0	d	d	d	1	0	0	0

byte 3: rel8
byte 4: #data8

CJNE [Rd], #data16, rel8

Bytes: 5
Cycles: 10t/7nt

Encoding:

1	1	1	0	1	0	1	1

0	d	d	d	1	0	0	0

byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CLR Clear Bit

Syntax: CLR bit

Operation: (bit) <-- 0

Description: Writes a 0 (clears) to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	0	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

CMP Integer Compare

Syntax: dest, src

Operation: dest - src

Description: The byte or word specified by the source operand is compared to the specified destination operand by performing a twos complement binary subtraction of src from dest. The flags are set according to the rules of subtraction. The source and destination data are not affected by the operation.

Size: byte-byte, word-word

Flags Updated: C, AC, V, N, Z

CMP Rd, Rs

Operation: (Rd) - (Rs)

Bytes: 2
Cycles: 3

Encoding:

0	1	0	0	DS	0	0	1

d	d	d	d	s	s	s	s

CMP Rd, [Rs]

Operation: (Rd) - ((WS:Rs))

Bytes: 2
Cycles: 4

Encoding:

0	1	0	0	DS	0	1	0

d	d	d	d	0	s	s	s

CMP [Rd], Rs

Operation: ((WS:Rd)) - (Rs)

Bytes: 2
Cycles: 4
Encoding:

0	1	0	0	DS	0	1	0

s	s	s	s	1	d	d	d

CMP Rd, [Rs+offset8]Bytes:

Cycles: 6
Operation: (Rd) - ((WS:Rs)+offset8)
Encoding:

0	1	0	0	DS	1	0	0

d	d	d	d	0	s	s	s

byte 3: offset8

CMP [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) - (Rs)
Encoding:

0	1	0	0	DS	1	0	0

s	s	s	s	1	d	d	d

byte 3: offset8

CMP Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) - ((WS:Rs)+offset16)
Encoding:

0	1	0	0	DS	1	0	1

d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) - (Rs)
Encoding:

0	1	0	0	DS	1	0	1	s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	1	0	0	DS	0	1	1	d	d	d	d	0	s	s	s

CMP [Rd+], Rs

Bytes: 2
Cycles: 5

Operation: ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:

0	1	0	0	DS	0	1	1	s	s	s	s	1	d	d	d

CMP direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) - (Rs)
Encoding:

0	1	0	0	DS	1	1	0	s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

CMP Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) - (direct)
Encoding:

0	1	0	0	DS	1	1	0

d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

CMP Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) - #data8
Encoding:

1	0	0	1	0	0	0	1

d	d	d	d	0	1	0	0

byte 3: #data8

CMP Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) - #data16
Encoding:

1	0	0	1	1	0	0	1

d	d	d	d	0	1	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) - #data8
Encoding:

1	0	0	1	0	0	1	0

0	d	d	d	0	1	0	0

byte 3: #data8

CMP [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) - #data16
Encoding:

1	0	0	1	1	0	1	0

0	d	d	d	0	1	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:

1	0	0	1	0	0	1	1

0	d	d	d	0	1	0	0

byte 3: #data8

CMP [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1

0	d	d	d	0	1	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) - #data8
Encoding:

1	0	0	1	0	1	0	0

0	d	d	d	0	1	0	0

byte 3: offset8
byte 4: #data8

CMP [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) - #data16
Encoding:

1	0	0	1	1	1	0	0

0	d	d	d	0	1	0	0

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CMP [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) - #data8
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	0	1	0	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

CMP [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) - #data16
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	0	1	0	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

CMP direct, #data8

Bytes: 4
Cycles: 4
Operation: (direct) - #data8
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	0	1	0	0

byte 3: lower 8 bits of direct
byte 4: #data8

CMP direct, #data16

Bytes: 5
Cycles: 4
Operation: (direct) - #data16
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	1	0	0

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CPL Integer Ones Complement

Syntax: CPL Rd

Operation: Rd <-- ($\overline{Rd}$)

Description: Performs a ones complement of the destination operand specified by the register Rd. The result is stored back into Rd. The destination may be either a byte or a word.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Cycles: 3

Encoding:

1	0	0	1	DS	0	0	0

d	d	d	d	1	0	1	0

DA Decimal Adjust

Syntax: DA Rd

Operation:

```
if (Rd.3-0) > 9 or (AC) = 1
        then (Rd.3-0) <-- (Rd.3-0) + 6
if (Rd.7-4) > 9 or (C) = 1
        then (Rd.7-4) <-- (Rd.7-4) + 6
```

Description: Adjusts the destination register to BCD format (binary-coded decimal) following an ADD or ADDC operation on BCD values. This operation may only be done on a byte register.

If the lower 4 bits of the destination value are greater than 9, or if the AC flag is set, 6 is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value.

If the upper 4 bits of the destination value are greater than 9, or if the carry flag was set by the add to the lower bits, 60 hex is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value. Carry will never be cleared by the DA instruction if it was already set.

Size: Byte

Flags Updated: C, N, Z

The carry flag may be set but not cleared. See the description of the carry flag update above.

Bytes: 2
Cycles: 4

Encoding:

1	0	0	1	0	0	0	0

d	d	d	d	1	0	0	0

Note: Please refer to the table on the next page.

The following table shows the possible actions that may occur during the DA instruction, related to the input conditions.

Table 6.6

Low nibble (bits 3-0)	AC	Carry to high nibble	High nibble (bits 7-4)	Initial C flag	Number added to value	Resulting C flag
0 - 9	0	0	0 - 9	0	00	0
A - F	0	1	0 - 8	0	06	0
0 - 3 *	1	0	0 - 9	0	06	0
0 - 9	0	0	A - F	0	60	1
A - F	0	1	9 - F	0	66	1
0 - 3 *	1	0	A - F	0	66	1
0 - 9	0	0	0 - 2 **	1	60	1
A - F	0	1	0 - 2 **	1	66	1
0 - 3 *	1	0	0 - 3 ***	1	66	1

: The largest digit that could result from adding two BCD digits that caused the AC flag to be set is 3. This is with an ADDC instruction where 9 + 9 + 1 (the carry flag) = 13 hex.

** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with no carry from the bottom nibble (the AC flag = 0) is 2. for instance, 98 hex + 97 hex = 12F hex.

*** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with a carry from the bottom nibble (the AC flag = 1) is 3. For instance, 99 hex + 99 hex = 132 hex.

DIV.w	**16x8**	**Signed Division**
DIV.d	**32x16**	**Signed Division**
DIVU.b	**8x8**	**Unsigned Division**
DIVU.w	**16x8**	**Unsigned Division**
DIVU.d	**32x16**	**Unsigned Division**

Description: The byte or word specified by the source operand is divided into the variable specified by the destination operand.

For DIVU.b, the destination operand can be any byte register that is the least significant byte of a word register. For DIV.w and DIVU.w, the destination operand must be a word register, and for DIV.d and DIVU.d, the destination operand must identify a word register that is the low-word of a double-word register (see note below). The result is stored in the destination register as the quotient (8 bits for DIVU.b, DIVU.w, DIV.w, and DIVU.w, and 16-bits for DIV.d and DIVU.d) in the least significant half and the remainder (same size as the quotient), in the most significant half (except for DIVU.b which stores the quotient in the destination as identified by the lower half of a word register and the remainder at upper half of the same word register).

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte-Byte, Word-Byte, Double word-Word

Flags Updated: C, V, N, Z

The carry flag is always cleared. The V flag is set in the following cases, otherwise it is cleared:

- DIVU.b: V is set if a divide by 0 occurred. A divide by 0 also causes a hardware trap to be generated.
- DIV.w, DIVU.w: V is set if the result of the divide is larger than 8 bits (the result does not fit in the destination).
- DIV.d, DIVU.d: V is set if the result of the divide is larger than 16 bits (the result does not fit in the destination).

The Z, and N flags are set based on the quotient (integer) portion of the result only and not on the remainder.

Examples:

a) DIVU.b R4L, R4H - will store the result of the division of R4L by R4H in R4L and R4H (quotient in register R4L, remainder in register R4H).

b) DIV.w R0, R2L - will store the result of word register R0 divided by byte register R2L in word register R0 (quotient in register R0L, remainder in register R0H).

c) DIV.d R4,R2 - will store the result of double-word register R5:R4 divided by word register R2 in double-word register R5:R4 (quotient in R4, remainder in R5)

Note: For all divides except DIVU.b, the destination register size is the same as indicated by the instruction (by the ".b", ".w", or ".d") and the source register is half that size.

DIV.w Rd, Rs
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Cycles: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:

1	1	1	0	0	1	1	1		d	d	d	d	s	s	s	s

DIV.w Rd, #data8
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 3
Cycles: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:

1	1	1	0	1	0	0	0		d	d	d	d	1	0	1	1

byte 3: #data8

DIV.d Rd, Rs
(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Cycles: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (signed divide)
(Rd+1)<-- 16-bit remainder of (Rd) / (Rs)
Encoding:

1	1	1	0	1	1	1	1		d	d	d	0	s	s	s	s

DIV.d Rd, #data16

(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 4
Cycles: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (signed divide)
(Rd+1)<-- 16-bit remainder of (Rd) / #data16
Encoding:

1	1	1	0	1	0	0	1		d	d	d	0	1	0	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

DIVU.b Rd, Rs

(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Cycles: 12
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:

1	1	1	0	0	0	0	1		d	d	d	d	s	s	s	s

DIVU.b Rd, #data8

(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 3
Cycles: 12
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:

1	1	1	0	1	0	0	0		d	d	d	d	0	0	0	1

byte 3: #data8

DIVU.w Rd, Rs

(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Cycles: 12
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:

1	1	1	0	0	1	0	1

d	d	d	d	s	s	s	s

DIVU.w Rd, #data8

(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 3
Cycles: 12

Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:

1	1	1	0	1	0	0	0

d	d	d	d	0	0	1	1

byte 3: #data8

DIVU.d Rd, Rs

(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Cycles: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (unsigned divide)
(Rd+1)<-- 16-bit remainder of (Rd) / (Rs)
Encoding:

1	1	1	0	1	1	0	1

d	d	d	0	s	s	s	s

DIVU.d Rd, #data16
(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 4
Cycles: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (unsigned divide)
(Rd+1)<-- 16-bit remainder of (Rd) / #data16
Encoding:

1	1	1	0	1	0	0	1		d	d	d	0	0	0	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

DJNZ Decrement and jump if not zero

Syntax: DJNZ dest, rel8

Operation:
(PC) <-- (PC) + 3
(dest) <-- (dest) - 1
if (Z) = 0 then
(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: Controls a loop of instructions. The parameters are: a condition code (Z), a counter (register or memory), and a displacement value. The instruction first decrements the counter by one, tests the condition if the result of decrement is 0 (for termination of the loop); if it is false, execution continues with the next instruction. If true, execution branches to the location indicated by the current value of the PC plus the sign extended displacement. The value in the PC is the address of the instruction following DJNZ.

The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.The destination operand could be byte or word.

Note: Refer to section 6.3 for details of jump range

Size: Byte, Word

Flags Updated: N, Z

DJNZ Rd, rel8

Bytes: 3
Cycles: 8t/5nt
Encoding:

1	0	0	0	DS	1	1	1

d	d	d	d	1	0	0	0

byte 3: rel8

DJNZ direct, rel8

Bytes: 4
Cycles: 9t/5nt
Encoding:

1	1	1	0	DS	0	1	0

0	0	0	0	1	direct: 3 bits

byte 3: lower 8 bits of direct
byte 4: rel8

FCALL Far Call Subroutine Absolute

Syntax: FCALL addr24

Operation:
(PC) <-- (PC) + 4
(SP) <-- (SP) - 4
((SP)) <-- (PC)
(PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space. The 24-bit return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned as CALL or branch will force PC.bit0 to 0.

Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 4
Cycles: 12/9(PZ)

Encoding:

1	1	0	0	0	1	0	0	address: middle 8 bits (bits 15-8)

byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

FJMP Far Jump Absolute

Syntax: FJMP addr24

Operation: (PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space.
Note: The target address must be word aligned as JMP always forces PC to an even address.

Note: if the XA is in page 0 mode, only 16-bits of the address will be used.

Size: None

Flags Updated: none

Bytes: 4
Cycles: 7

Encoding:

1	1	0	1	0	1	0	0		address: middle 8 bits (bits 15-8)

byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

JB Relative Jump if bit set

Syntax: JB bit, rel8

Operation:

```
(PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0
```

Description: If the specified bit is a one, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is clear, the instruction following JB is executed.The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Cycles: 6t/3nt

Encoding:

1	0	0	1	0	1	1	1

1	0	0	0	0	0	bit: 2

byte 3: lower 8 bits of bit address
byte 4: rel8

JBC Jump if bit is set then clear bit

Syntax: JBC bit, rel8

Operation:

```
(PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0; (bit) <-- 0
```

Description: If the bit specified is set, branch to the address pointed to by the PC plus the specified displacement. The specified bit is then cleared allowing implementation of semaphore operations. If the specified bit is clear, the instruction following JBC is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Cycles: 6t/3nt

Encoding:

1	0	0	1	0	1	1	1

1	1	0	0	0	0	bit: 2

byte 3: lower 8 bits of bit address
byte 4: rel8

JMP Relative Jump

Syntax: JMP rel16

Operation: (PC) <-- (PC) + 3
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Jumps unconditionally. The branch range is +65,535 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: None

Flags Updated: none

Bytes: 3
Cycles: 6

Encoding:

1	1	0	1	0	1	0	1		rel16: upper 8 bits

byte 3: lower 8 bits of rel16

JMP Jump Indirect through Register

Syntax: JMP [Rs]

Operation:
(PC) <-- (PC) + 2
(PC.15-1) <-- (Rs.15-1) (note that PC.23-16 is not affected)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address contained in the operand word register, anywhere within the 64K segment following the JMP instruction.The value of the PC used in the target address calculation is the address of the instruction following the JMP instruction. The target address must be word aligned as JMP will force PC.bit0 to 0.

Size: none

Flags Updated: none

Bytes: 2
Cycles: 7

Encoding:

1	1	0	1	0	1	1	0		0	1	1	1	0	s	s	s

JMP Jump indirect through register

Syntax: JMP [A+DPTR]

Operation: (PC) <-- (PC) + 2
(PC15-1) <-- (A) + (DPTR)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address formed by the sum of the 80C51 compatibility registers A and DPTR, anywhere within the 64K segment following the JMP instruction. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Note: The target address must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Cycles: 5

Note: A and DPTR are pre-defined registers used for 80C51 code translation.

Encoding:

1	1	0	1	0	1	1	0

0	1	0	0	0	1	1	0

JMP Jump double indirect

Syntax: JMP [[Rs+]]

Operation:
(PC) <-- (PC) + 2
(PC.15-0) <-- code memory ((WS:Rs))
(PC.0) <-- 0
(Rs) <-- (Rs) + 2

Description: Causes an unconditional branch to the address contained in memory at the address pointed to by the register specified in the instruction. The specified register is post-incremented.

This 2-byte instruction may be used to compress code size by using it to index through a table of procedure addresses that are accessed in sequence. Each procedure would end with another JMP [[R+]] that would immediately go to the next procedure whose address is in the table.

The procedures should, however, must be located in the same 64K address page of the executed "Jump Double-indirect" instruction (although the table could be in any page). This results in substantial code compression and hence cost reduction through smaller memory requirements. The register pointer (index to the table) being automatically post-incremented after the execution of the instruction. The 24-bit address is identified by combining the low order 16-bit of the PC and either of high 8-bits of PC or the contents of a byte-size CS register as chosen by the program through a segment select Special Function Register (SFR).

Note: The subroutine addresses must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Cycles: 5

Encoding:

1	1	0	1	0	1	1	0

0	1	1	0	0	s	s	s

JNB Jump if bit not set

Syntax: JNB bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 0 then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: If the specified bit is a zero, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is set, the instruction following JB is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Cycles: 6t/3nt

Encoding:

1	0	0	1	0	1	1	1

1	0	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address
byte 4: rel8

JNZ Jump if the A register is not zero

Syntax: JNZ rel8

Operation: (PC) <-- (PC) + 2
if (A) not equal to 0, then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: A relative branch is taken if the contents of the 80C51 Accumulator are not zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	0	1	1	1	0	rel8

JZ Jump if the A register is zero

Syntax: JZ rel8

Operation:

```
(PC) <-- (PC) + 2
If (A) = 0 then
(PC.15-0) <-- (PC + rel8*2);
(PC.0) <-- 0
```

Description: A relative branch is taken if the contents of the 80C51 Accumulator are zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Cycles: 6t/3nt

Encoding:

1	1	1	0	1	1	0	0	rel8

LEA Load effective address

Syntax: LEA Rd, Rs+offset8/16

Operation: (Rd) <-- (Rs)+offset8/16

Description: The word specified by the source operand is added to the offset value and the result is stored into the register specified by the destination operand. The source and destination operands are both registers. The offset value is an immediate data field of either 8 or 16 bits in length. The source data is not affected by the operation.

This instruction mimics the address calculation done during other instructions when the register indirect with offset addressing mode is used, allowing the resulting address to be saved for other purposes.

Note: The result of this operation is always a word since it duplicates the calculation of the indirect with offset addressing mode.

Size: Word-Word

Flags Updated: none

LEA Rd, Rs+offset8

Bytes: 3
Cycles: 3
Encoding:

0	1	0	0	0	0	0	0	0	d	d	d	0	s	s	s

byte 3: offset8

LEA Rd, Rs+offset16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rs)+offset16
Encoding:

0	1	0	0	1	0	0	0	0	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

LSR Logical Shift Right

Syntax: LSR dest, count

Operation:

```
Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
(dest.msb) <- 0
count = count-1
End While
```

Description: If the count operand is greater than the variable specified by the destination operand is logically shifted right by the number of bits specified by the count operand. The MSBs of the result are filled with zeroes.The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.The count operand is a positive value which may be from 0 to 31. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register. The count is not affected by the operation.

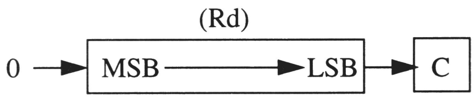
Note:
- For Logical Shift Left, use ASL ignoring the N flag.
- If shift count (count in Rs) exceeds data size, the shifting is truncated to 5 bits i.e 32 else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte, Word, Double Word

Flags Updated: C, N, Z (N = 0 after an LSR)

LSR Rd, Rs (Rs = Byte-register)

Operation:

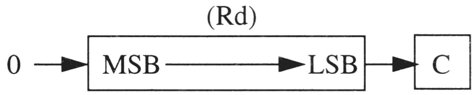


Bytes: 2
Cycles: 4 + 1 for each 2 bits of shift
Encoding:

1	1	0	0	DS1	DS0	0	0	d	d	d	d	s	s	s	s

LSR Rd, #data4
Rd, #data5

Operation:



Bytes:	2
Cycles:	6 + 1 for each 2 bits of shift

Encoding: (for byte and word data sizes)

1	1	0	1	DS1	DS0	0	0

d	d	d	d	#data4

(for double word data size)

1	1	0	1	1	1	0	0

d	d	d	#data5

Note: DS1/DS0 = 00: byte operation; DS1/DS0 = 01: reserved; DS1/DS0 = 10: word operation; DS1/DS0 = 11: double word operation.

MOV Move Data

Syntax: MOV dest, src

Operation: dest <- src

Description: The byte or word specified by the source operand is copied into the variable specified by the destination operand. The source data is not affected by the operation.

Source and destination operands may be a register in the register file, an indirect address specified by a pointer register, an indirect address specified by a pointer register added to an immediate offset of 8 or 16 bits, or a direct address. Source operands may also be specified as immediate data contained within the instruction. Auto-increment of the indirect pointers is available for simple indirect (not offset) addressing.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

MOV Rd, Rs

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rs)
Encoding:

1	0	0	0	DS	0	0	1

d	d	d	d	s	s	s	s

MOV Rd, [Rs]

Bytes: 2
Cycles: 3
Operation: (Rd) <-- ((WS:Rs))
Encoding:

1	0	0	0	DS	0	1	0

d	d	d	d	0	s	s	s

MOV [Rd], Rs

Bytes: 2
Cycles: 3

Operation: ((WS:Rd)) <-- (Rs)
Encoding:

1	0	0	0	DS	0	1	0		s	s	s	s	1	d	d	d

MOV Rd, [Rs+offset8]

Bytes: 3
Cycles: 5
Operation: (Rd) <-- ((WS:Rs)+offset8)
Encoding:

1	0	0	0	DS	1	0	0		d	d	d	d	0	s	s	s

byte 3: offset8

MOV [Rd+offset8], Rs

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)+offset8) <-- (Rs)
Encoding:

1	0	0	0	DS	1	0	0		s	s	s	s	1	d	d	d

byte 3: offset8

MOV Rd, [Rs+offset16]

Bytes: 4
Cycles: 5
Operation: (Rd) <-- ((WS:Rs)+offset16)
Encoding:

1	0	0	0	DS	1	0	1		d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV [Rd+offset16], Rs

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)+offset16) <-- (Rs)

Encoding:

1	0	0	0	DS	1	0	1
s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV Rd, [Rs+]

Bytes: 2
Cycles: 4
Operation: (Rd) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:

1	0	0	0	DS	0	1	1
d	d	d	d	0	s	s	s

MOV [Rd+], Rs

Bytes: 2
Cycles: 4

Operation: ((WS:Rd)) <-- (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:

1	0	0	0	DS	0	1	1
s	s	s	s	1	d	d	d

MOV [Rd+], [Rs+]

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:

1	0	0	1	DS	0	0	0
0	d	d	d	0	s	s	s

MOV direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) <-- (Rs)
Encoding:

1	0	0	0	DS	1	1	0

s	s	s	s	1	direct:3 bits

byte 3: lower 8 bits of direct

MOV Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (direct)
Encoding:

1	0	0	0	DS	1	1	0

d	d	d	d	0	direct:3 bits

byte 3: lower 8 bits of direct

MOV direct, [Rs]

Bytes: 3
Cycles: 4
Operation: (direct) <-- ((WS:Rd))
Encoding:

1	0	1	0	DS	0	0	0

1	s	s	s	0	direct:3 bits

byte 3: lower 8 bits of direct

MOV [Rd], direct

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- (direct)
Encoding:

1	0	1	0	DS	0	0	0

0	d	d	d	0	direct:3 bits

byte 3: lower 8 bits of direct

MOV Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- #data8
Encoding:

1	0	0	1	0	0	0	1

d	d	d	d	1	0	0	0

byte 3: #data8

MOV Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- #data16
Encoding:

1	0	0	1	1	0	0	1

d	d	d	d	1	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd], #data8

Bytes: 3
Cycles: 3
Operation: ((WS:Rd)) <-- #data8
Encoding:

1	0	0	1	0	0	1	0

0	d	d	d	1	0	0	0

byte 3: #data8

MOV [Rd], #data16

Bytes: 4
Cycles: 3
Operation: ((WS:Rd)) <-- #data16
Encoding:

1	0	0	1	1	0	1	0

0	d	d	d	1	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd+], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- #data8
(Rd) <-- (Rd) + 1
Encoding:

1	0	0	1	0	0	1	1		0	d	d	d	1	0	0	0

byte 3: #data8

MOV [Rd+], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- #data16
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1		0	d	d	d	1	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd+offset8], #data8

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)+offset8) <-- #data8
Encoding:

1	0	0	1	0	1	0	0		0	d	d	d	1	0	0	0

byte 3: offset8
byte 4: #data8

MOV [Rd+offset8], #data16

Bytes: 5
Cycles: 5
Operation: ((WS:Rd)+offset8) <-- #data16
Encoding:

1	0	0	1	1	1	0	0		0	d	d	d	1	0	0	0

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV [Rd+offset16], #data8

Bytes: 5
Cycles: 5
Operation: ((WS:Rd)+offset16) <-- #data8
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	1	0	0	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

MOV [Rd+offset16], #data16

Bytes: 6
Cycles: 5
Operation: ((WS:Rd)+offset16) <-- #data16
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	1	0	0	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

MOV direct, #data8

Bytes: 4
Cycles: 3
Operation: (direct) <-- #data8
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	1	0	0	0

byte 3: lower 8 bits of direct
byte 4: #data8

MOV direct, #data16

Bytes: 5
Cycles: 3
Operation: (direct) <-- #data16
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	1	0	0	0

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV direct, direct

Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct)
Encoding:

1	0	0	1	DS	1	1	1

0	d dir: 3 bits	0	s dir: 3 bits

byte 3: lower 8 bits of direct (dest)
byte 4: lower 8 bits of direct (src)

MOV Rd, USP (move from user stack pointer)

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (USP)
Encoding:

1	0	0	1	0	0	0	0

d	d	d	d	1	1	1	1

MOV USP, Rs (move to user stack pointer)

Bytes: 2
Cycles: 3
Operation: (USP) <-- (Rs)
Encoding:

1	0	0	1	1	0	0	0

s	s	s	s	1	1	1	1

MOV Move Bit to Carry

Syntax: MOV C, bit

Operation: (C) <-- (bit)

Description: Copies the specified bit to the carry flag.

Size: Bit

Flags Updated: none

Note: C is written as the destination of the move, not as a status flag

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

MOV **Move Carry to Bit**

Syntax: MOV bit, C

Operation: (bit) <-- (C)

Description: Copies the carry flag to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	1	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

MOVC Move Code

Syntax: MOVC Rd, [Rs+]

Operation: (Rd) <-- code memory ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Description: Contents of code memory are copied to (or from[6])an internal register. The byte or word specified by the source operand is copied to the variable specified by the destination operand. In the case of MOVC, the pointer segment selection gives the choices of PC_{23-16} or CS segment (current *working segment* referred here as WS), rather than DS or ES as is used for all other instructions.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

Bytes: 2
Cycles: 4

Encoding:

1	0	0	0	DS	0	0	0

d	d	d	d	0	s	s	s

---

[6] Could be present in some XA derivatives with writable code memory like Flash etc.

MOVC Move Code to A (DPTR)

Syntax: MOVC A, [A+DPTR]

Operation: PC <- PC+2
(A) <-- code memory (PC.23-16:(A) + (DPTR))

Description: The byte located at the code memory address formed by the sum of A and the DPTR is copied to the A register. The A and DPTR registers are pre-defined registers used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Cycles: 6

Encoding:

1	0	0	1	0	0	0	0		0	1	0	0	1	1	1	0

MOVC Move Code to A (PC)

Syntax: MOVC A, [A+PC]

Operation: PC <- PC+2
(A) <-- code memory [PC.23-16: (A +PC.15-0)]

Note: Only 16-bits of A+PC are used

Description: The byte located at the code memory address formed by the sum of A and the current Program Counter value is copied to the A register. The A register is a pre-defined register used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Cycles: 6

Encoding:

1	0	0	1	0	0	0	0

0	1	0	0	1	1	0	0

MOVS Move Short

Syntax: MOVS dest, #data

Description: Four bits of signed immediate data are moved to the destination. The immediate data is sign-extended to the proper size, then moved to the variable specified by the destination operand, which may be a byte or a word. The immediate data range is +7 to -8. This instruction is used to save time and code space for the many instances where a small data constant is moved to a destination.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

MOVS Rd, #data4

Bytes: 2
Cycles: 3
Operation: (Rd) <-- sign-extended #data4
Encoding:

1	0	1	1	DS	0	0	1

d	d	d	d	#data4

MOVS [Rd], #data4

Bytes: 2
Cycles: 3
Operation: ((WS:Rd)) <-- sign-extended #data4
Encoding:

1	0	1	1	DS	0	1	0

0	d	d	d	#data4

MOVS [Rd+], #data4

Bytes: 2
Cycles: 4
Operation: ((WS:Rd)) <-- sign-extended #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

1	0	1	1	DS	0	1	1

0	d	d	d	#data4

MOVS [Rd+offset8], #data4

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)+offset8) <-- sign-extended #data4
Encoding:

1	0	1	1	DS	1	0	0

0	d	d	d	#data4

byte 3: offset8

MOVS [Rd+offset16], #data4

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)+offset16) <-- sign-extended #data4
Encoding:

1	0	1	1	DS	1	0	1

0	d	d	d	#data4

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOVS direct, #data4

Bytes: 3
Cycles: 3
Operation: (direct) <-- sign-extended #data4

Encoding:

1	0	1	1	DS	1	1	0

0	direct: 3 bits	#data4

byte 3: lower 8 bits of direct

MOVX Move External Data

Syntax: MOVX dest, src

Description: Move external data to or from an internal register. The byte or word specified by the source operand is copied into the variable specified by the destination operand. This instruction allows access to data external to the microcontroller in the address range of 0 to 64K. The standard indirect move may access external data only above the boundary where internal data RAM ends, whereas MOVX always forces an external access. MOVX only operates on the first 64K of external data memory. This instruction is included to allow compatibility with 80C51 code.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

MOVX Rd, [Rs]

Bytes: 2
Cycles: 6
Operation: (Rd) <-- external data memory ((Rs))

Encoding:

1	0	1	0	DS	1	1	1

d	d	d	d	0	s	s	s

MOVX [Rd], Rs

Bytes: 2
Cycles: 6
Operation: external data memory ((Rd)) <-- (Rs)
Encoding:

1	0	1	0	DS	1	1	1

s	s	s	s	1	d	d	d

MUL.w 16x16 Signed Multiply
MULU.b 8x8 Unsigned Multiply
MULU.w 16x16 Unsigned Multiply

Description: The byte or word specified by the source operand is multiplied by the variable specified by the destination operand.

The destination operand must be the first half of a double size register (word for a byte multiply and double word for a word multiply). The result is stored in the double size register.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, and R7:R6).

Size: Byte-Byte, Word-Word

Flags Updated: C, V, N, Z

The carry flag is always cleared by a multiply instruction. The V flag is set in the following cases, otherwise it is cleared:
- MULU.b: V is set if the result of the multiply is greater than FFh (the upper byte is not equal to 0).
- MULU.w: V is set if the result of the multiply is greater than FFFFh (the upper word is not equal to 0).
- MUL.w: V is set if the absolute value of the result of the multiply is greater than 7FFFh (the upper word is not a sign extension of the lower word).

Examples:
a) MUL.w R0,R5 stores the product of word register 0 and word register 5 in double word register 0 (least significant word in word register R0, most significant word in word register R1).

b) MULU.b R4L, R4H will store the MS byte of the product of R4L and R4H in R4H and the LS byte in R4L.

MUL.w Rd, Rs

(signed 16 bits * 16 bits --> 32 bits)

Bytes: 2
Cycles: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * (Rs) (signed multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:

1	1	1	0	0	1	1	0

d	d	d	d	s	s	s	s

MUL.w Rd, #data16

(signed 16 bits * 16 bits --> 32 bits)

Bytes: 4
Cycles: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * #data16 (signed multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:

1	1	1	0	1	0	0	1

d	d	d	d	1	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MULU.b Rd, Rs

(unsigned 8 bits * 8 bits --> 16 bits)

Bytes: 2
Cycles: 12
Operation: (RdH) <-- Most significant byte of (Rd) * (Rs) (unsigned multiply)
(RdL) <-- Least significant byte of (Rd) * (Rs)
Encoding:

1	1	1	0	0	0	0	0

d	d	d	d	s	s	s	s

MULU.b Rd, #data8
(unsigned 8 bits * 8 bits --> 16 bits)

Bytes: 3
Cycles: 12
Operation: (RdH) <-- Most significant byte of (Rd) * #data8 (unsigned multiply)
(RdL) <-- Least significant byte of (Rd) * #data8
Encoding:

1	1	1	0	1	0	0	0		d	d	d	d	0	0	0	0

byte 3: #data8

MULU.w Rd, Rs
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 2
Cycles: 12
Operation: (Rd+1)<-- Most significant word of (Rd) * (Rs) (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:

1	1	1	0	0	1	0	0		d	d	d	d	s	s	s	s

MULU.w Rd, #data16
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 4
Cycles: 12
Operation: (Rd+1)<-- Most significant word of (Rd) * #data16 (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:

1	1	1	0	1	0	0	1		d	d	d	d	0	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

NEG Negate

Syntax: NEG Rd

Operation: Rd <-- ($\overline{\text{Rd}}$) + 1

Description: The destination register is negated (twos complement). The destination may be a byte or a word.

Size: Byte, Word

Flags Updated: V, N, Z

The V flag is set if a twos complement overflow occurred: the original value = result = 8000 hex for a word operation or 80 hex for a byte operation.

Bytes: 2
Cycles: 3

Encoding:

1	0	0	1	DS	0	0	0

d	d	d	d	1	0	1	1

NOP No Operation

Syntax: NOP

Operation: PC <- PC + 1

Description: Execution resumes at the following instruction. This instruction is defined as being one byte in length in order to allow it to be used to force word alignment of instructions that are branch targets, or for any other purpose. It may also be used to as a delay for a predictable amount of time.

Size: None

Flags Updated: none

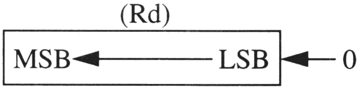
Bytes: 1
Cycles: 3

Encoding:

0	0	0	0	0	0	0	0

Syntax: NORM Rd, Rs

Operation:



Description: Logically shifts left the contents of the destination until the MSB is set, storing the number of shifts performed in the count (source) register. The data size may be 8, 16, or 32 bits.

If the destination value already has the MSB set, the count returned will be 0. If the destination value is 0, the count returned will be 0, the N flag will be cleared, and the Z flag will be set. For all other conditions, the N flag will be 1 and the Z flag will be 0.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
The last pair, i.e, R7:R6 is probably not a good idea as R7 is the current stack pointer.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Cycles: For 8 or 16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding:

1	1	0	0	DS1	DS0	1	1	d	d	d	d	s	s	s	s

Note: DS/DS = 00: byte operation; DS/DS = 01: reserved; DS/DS = 10: word operation; DS/DS = 11: double word operation.

OR **Logical OR**

Syntax: OR dest, src

Description: Bitwise logical OR the contents of the source to the destination. The byte or word specified by the source operand is logically ORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

OR Rd, Rs

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rd) + (Rs)
Encoding:

0	1	1	0	DS	0	0	1

d	d	d	d	s	s	s	s

OR Rd, [Rs]

Bytes: 2
Cycles: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:

0	1	1	0	DS	0	1	0

d	d	d	d	0	s	s	s

OR [Rd], Rs

Bytes: 2
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
Encoding:

0	1	1	0	DS	0	1	0

s	s	s	s	1	d	d	d

OR Rd, [Rs+offset8]

Bytes: 3
Cycles: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:

0	1	1	0	DS	1	0	0

d	d	d	d	0	s	s	s

byte 3: offset8

OR [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:

0	1	1	0	DS	1	0	0

s	s	s	s	1	d	d	d

byte 3: offset8

OR Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:

0	1	1	0	DS	1	0	1

d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

OR [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:

0	1	1	0	DS	1	0	1

s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

OR Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	1	1	0	DS	0	1	1

d	d	d	d	0	s	s	s

OR [Rd+], Rs

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	1	1	0	DS	0	1	1

s	s	s	s	1	d	d	d

OR direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:

0	1	1	0	DS	1	1	0

s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

OR Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:

0	1	1	0	DS	1	1	0

d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

OR Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:

1	0	0	1	0	0	0	1		d	d	d	d	0	1	1	0

byte 3: #data8

OR Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:

1	0	0	1	1	0	0	1		d	d	d	d	0	1	1	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:

1	0	0	1	0	0	1	0		0	d	d	d	0	1	1	0

byte 3: #data8

OR [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:

1	0	0	1	1	0	1	0		0	d	d	d	0	1	1	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:

1	0	0	1	0	0	1	1		0	d	d	d	0	1	1	0

byte 3: #data8

OR [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1		0	d	d	d	0	1	1	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:

1	0	0	1	0	1	0	0		0	d	d	d	0	1	1	0

byte 3: offset8
byte 4: #data8

OR [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:

1	0	0	1	1	1	0	0		0	d	d	d	0	1	1	0

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

OR [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	0	1	1	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

OR [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	0	1	1	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

OR direct, #data8

Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct) + #data8
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	0	1	1	0

byte 3: lower 8 bits of direct
byte 4: #data8

OR direct, #data16

Bytes: 5
Cycles: 4
Operation: (direct) <-- (direct) + #data16
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	1	1	0

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ORL Logical OR bit

Syntax: ORL C, bit

Operation:(C) <-- (C) + (bit)

Description: Logical (inclusive) OR a bit to the Carry flag. Read the specified bit and logically OR it to the Carry flag.
(C is written as the destination of the ORL, not as a status flag)

Size: Bit

Flags Updated: none

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

ORL Logical OR complement of bit

Syntax: ORL C, /bit

Operation: (C) <-- (C) + ($\overline{\text{bit}}$)

Description: Logically OR the complement of a bit to the Carry flag. Read the specified bit, complement it, and logically OR it to the Carry flag.
(C is written as the destination of the move, not as a status flag)

Flags Updated: none

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	1	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

POP Pop
POPU Pop User

Syntax: POP dest

Description: The stack is popped and the data written to the specified directly addressed location. The data size may be byte or word. POP uses the current stack pointer, while POPU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

POP direct

Bytes: 3
Cycles: 5
Operation: (direct) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:

1	0	0	0	DS	1	1	1

0	0	0	1	0	direct: 3 bits

byte 3: 8 bits of direct

POPU direct

Bytes: 3
Cycles: 5
Operation: (direct) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:

1	0	0	0	DS	1	1	1

0	0	0	0	0	direct: 3 bits

byte 3: 8 bits of direct

POP Pop Multiple
POPU Pop User Multiple

Syntax: POP Rlist
POPU Rlist

Description: Pop the specified registers (one or more) from the stack. The stack is popped (from 1 to 8 times) and the data stored in the specified registers. Any combination of word registers in the group R0 to R7 may be popped in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be popped in a single instruction in a byte operation. POP uses the current stack pointer, while POPU forces an access to the user stack.

Note: Rlist is a bit map that represents each register to be popped. The registers are in the order R7, R6, R5,......, R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The pop order is from right to left i.e the register specified by the rightmost one in Rlist will be popped first, etc. The order must be the reverse of that used by the preceding PUSH instruction.

Size: Byte, Word

Flags Updated: none

POP Rlist

Bytes: 2
Cycles: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:

0	H/L	1	0	DS	1	1	1		Rlist

POPU Rlist

Bytes: 2
Cycles: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:

0	H/L	1	1	DS	1	1	1		Rlist

PUSH **Push**
PUSHU **Push User**

Syntax: PUSH src
PUSHU src

Description: The specified directly addressed data is pushed onto the stack. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

PUSH direct

Bytes: 3
Cycles: 5
Operation: (SP) <-- (SP) - 2
((SP)) <-- (direct)
Encoding:

1	0	0	0	DS	1	1	1

0	0	1	1	0	direct: 3 bits

byte 3: 8 bits of direct

PUSHU direct

Bytes: 3
Cycles: 5
Operation: (USP) <-- (USP) - 2
((USP)) <-- (direct)
Encoding:

1	0	0	0	DS	1	1	1

0	0	1	0	0	direct: 3 bits

byte 3: 8 bits of direct

PUSH Push Multiple
PUSHU Push User Multiple

Syntax: PUSH Rlist
PUSHU Rlist

Description: Push the specified registers (one or more) onto the stack. The specified registers are pushed onto the stack. Any combination of word registers in the group R0 to R7 may be pushed in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be pushed in a single instruction in a byte operation. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack. PUSHU is only available to system mode code.

Note: Rlist is a bit map that represents each register to be popped. The registers are in the order R7, R6, R5,......, R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The pop order is from left to right i.e the register specified by the leftmost one in Rlist will be pushed first, etc. The order must be the reverse of that used by the corresponding POP instruction. This order results in the registers appearing in memory in the same order that they appear in the register file.

Size: Byte, Word

Flags Updated: none

PUSH Rlist

Bytes: 2
Cycles: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(SP) <-- (SP) - 2
((SP)) <-- (Ri)
Encoding:

0	H/L	0	0	DS	1	1	1	Rlist

PUSHU Rlist

Bytes: 2
Cycles: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(USP) <-- (USP) - 2
((USP)) <-- (Ri)
Encoding:

0	H/L	0	1	DS	1	1	1	Rlist

RESET Software Reset

Syntax: RESET

Operation:
(PC) <-- vector(0)
(PSW) <-- vector(0)
(SFRs) <-- reset values (refer to the description of reset for details)

Description: The chip is reset exactly as if the external hardware reset has been asserted with the exception that it does not sample inputs for configuration, e.g, $\overline{\text{EA}}$, BUSW etc. When a RESET instruction is executed, the chip is internally reset, but no external $\overline{\text{RESET}}$ pulse is generated. The above inputs which are latched during rising edge of a $\overline{\text{RESET}}$ pulse, hence does not affect the chip configuration.

Flags Updated: The entire PSW is set to the value specified in the reset vector.

Bytes: 2
Cycles: 19

Encoding:

1	1	0	1	0	1	1	0

0	0	0	1	0	0	0	0

RET Return from Subroutine

Syntax: RET

Operation: (PC) <-- ((SP))
(SP) <-- (SP) + 4

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value (PC_{23-0}). This instruction is used to return from a subroutine that was called with a CALL or Far Call (FCALL).

Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: none

Bytes: 2
Cycles: 8/6 (PZ)

Encoding:

1	1	0	1	0	1	1	0

1	0	0	0	0	0	0	0

RETI Return from Interrupt

Syntax: RETI

Operation: (PSW) <-- ((SSP))
(PC.23-0) <-- ((SSP))
(SSP) <-- (SSP) + 6

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value. The Program Status Word is also restored by being popped from the stack.

This instruction is a privileged instruction (limited to system mode) and is used to return from an interrupt/exception. An attempt to use RETI in user mode will generate a trap.

Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: All PSW bits are written by the POP of the PSW value in System mode. In User mode, the protected PSW bits are not altered.

Bytes: 2
Cycles: 10/8 (PZ)

Encoding:

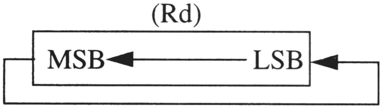
1	1	0	1	0	1	1	0

1	0	0	1	0	0	0	0

RL Rotate Left

Syntax: RL Rd, #data4

Operation:



```
count <- #data4
Do While (count not equal to 0)
    $(dest_0) \leftarrow (dest_{msb})$
    $(dest_n) \leftarrow (dest_{n-1})$
    (count) <- count -1
End While
```

Description: The variable specified by the destination operand is rotated left by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Cycles: 4 + 1 for each 2 bits of shift

Encoding:

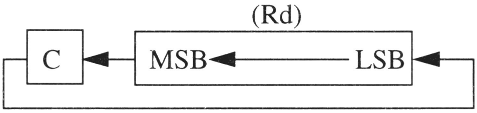
1	1	0	1	DS	0	1	1

d	d	d	d	#data4

RLC Rotate Left Through Carry

Syntax: RLC Rd, #data4

Operation:



```
count <- #data4
Do While (count not equal to 0)
(C) <- ($dest_{msb}$)
($dest_n$) <- ($dest_{n-1}$)
($dest_0$) <- (C)
(count) <- count -1
End While
```

Description: The variable specified by the destination operand is rotated left through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Cycles: 4 + 1 for each 2 bits of shift

Encoding:

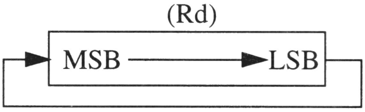
1	1	0	1	DS	1	1	1

d	d	d	d	#data4

RR **Rotate Right**

Syntax: RR Rd, #data4

Operation:



```
count <- #data4
Do While (count not equal to 0)
    $(dest_{msb})$ <- $(dest_0)$
    $(dest_{n-1})$ <- $(dest_n)$
    (count) <- count -1
End While
```

Description: If the count operand is greater than 0, the destination operand is rotated right by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15. If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Cycles: 4 + 1 for each 2 bits of shift

Encoding:

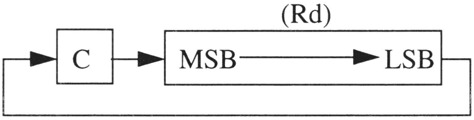
1	0	1	1	DS	0	0	0

d	d	d	d	#data4

RRC Rotate Right Through Carry

Syntax: RRC Rd, #data4

Operation:



```
count <- #data4
Do While (count not equal to 0)
$(C)$ <- $(dest_0)$
$(dest_n)$ <- $(dest_{n+1})$
$(dest_{msb})$ <- $(C)$
(count) <- count -1
End While
```

Description: If the count operand is greater than 0, the destination operand is rotated right through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.
If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Cycles: 4 + 1 for each 2 bits of shift

Encoding:

1	0	1	1	DS	1	1	1

d	d	d	d	#data4

SETB Set Bit

Syntax: SETB bit

Operation: (bit) <-- 1

Description: Writes (sets) a 1 to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Cycles: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	0	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

SEXT Sign Extend

Syntax: SEXT Rd

Operation:
```
if N = 1
        then (Rd) <-- FF in byte mode or FFFF in word mode
if N = 0
        then (Rd) <-- 00 in byte mode or 0000 in word mode
```

Description: Copies the N flag (the sign bit of the last ALU operation) into the destination register. The destination register may be a byte or word register.

Example:

SEXT.b R1
if the result of the previous operation left the N flag set, then R1 <-- FF

Size: Byte, word

Flags Updated: none

Bytes: 2
Cycles: 3

Encoding:

1	0	0	1	DS	0	0	0

d	d	d	d	1	0	0	1

SUB Integer Subtract

Syntax: SUB dest, src

Operation: dest <- dest - src

Description: Performs a twos complement binary subtraction of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

SUB Rd, Rs

Bytes:	2
Cycles:	3
Operation:	(Rd) <-- (Rd) - (Rs)
Encoding:	

0	0	1	0	DS	0	0	1

d	d	d	d	s	s	s	s

SUB Rd, [Rs]

Bytes:	2
Cycles:	4
Operation:	(Rd) <-- (Rd) - ((WS:Rs))
Encoding:	

0	0	1	0	DS	0	1	0

d	d	d	d	0	s	s	s

SUB [Rd], Rs

Bytes:	2
Cycles:	4
Operation:	((WS:Rd)) <-- ((WS:Rd)) - (Rs)
Encoding:	

0	0	1	0	DS	0	1	0

s	s	s	s	1	d	d	d

SUB Rd, [Rs+offset8]

Bytes: 3
Cycles: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8)
Encoding:

0	0	1	0	DS	1	0	0

d	d	d	d	0	s	s	s

byte 3: offset8

SUB [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs)
Encoding:

0	0	1	0	DS	1	0	0

s	s	s	s	1	d	d	d

byte 3: offset8

SUB Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16)
Encoding:

0	0	1	0	DS	1	0	1

d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUB [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs)
Encoding:

0	0	1	0	DS	1	0	1

s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUB Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	1	0	DS	0	1	1

d	d	d	d	0	s	s	s

SUB [Rd+], Rs

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	1	0	DS	0	1	1

s	s	s	s	1	d	d	d

SUB direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) <-- (direct) - (Rs)
Encoding:

0	0	1	0	DS	1	1	0

s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

SUB Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (Rd) - (direct)
Encoding:

0	0	1	0	DS	1	1	0

d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

SUB Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- (Rd) - #data8
Encoding:

1	0	0	1	0	0	0	1

d	d	d	d	0	0	1	0

byte 3: #data8

SUB Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rd) - #data16
Encoding:

1	0	0	1	1	0	0	1

d	d	d	d	0	0	1	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
Encoding:

1	0	0	1	0	0	1	0

0	d	d	d	0	0	1	0

byte 3: #data8

SUB [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
Encoding:

1	0	0	1	1	0	1	0

0	d	d	d	0	0	1	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:

1	0	0	1	0	0	1	1		0	d	d	d	0	0	1	0

byte 3: #data8

SUB [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1		0	d	d	d	0	0	1	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8
Encoding:

1	0	0	1	0	1	0	0		0	d	d	d	0	0	1	0

byte 3: offset8
byte 4: #data8

SUB [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16
Encoding:

1	0	0	1	1	1	0	0		0	d	d	d	0	0	1	0

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUB [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	0	0	1	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUB [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	0	0	1	0

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUB direct, #data8

Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct) - #data8
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	0	0	1	0

byte 3: lower 8 bits of direct
byte 4: #data8

SUB direct, #data16

Bytes: 5
Cycles: 4
Operation: (direct) <-- (direct) - #data16
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	0	1	0

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUBB Subtract with Borrow

Syntax: SUBB dest, src

Operation: dest <- dest - src - C

Description: Performs a twos complement binary addition of the source operand and the previously generated carry bit (borrow) with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is zero (C = 0, i.e, Borrow = 1), the result is exact difference of the operands; if it is one (C = 1, i.e, Borrow = 0), the result is 1 less than the difference in operands.

This form of subtraction is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then SUBB is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

SUBB Rd, Rs

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rd) - (Rs) - (C)
Encoding:

0	0	1	1	DS	0	0	1

d	d	d	d	s	s	s	s

SUBB Rd, [Rs]

Bytes: 2
Cycles: 4
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
Encoding:

0	0	1	1	DS	0	1	0

d	d	d	d	0	s	s	s

SUBB [Rd], Rs

Bytes: 2
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
Encoding:

0	0	1	1	DS	0	1	0		s	s	s	s	1	d	d	d

SUBB Rd, [Rs+offset8]

Bytes: 3
Cycles: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8) - (C)
Encoding:

0	0	1	1	DS	1	0	0		d	d	d	d	0	s	s	s

byte 3: offset8

SUBB [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs) - (C)
Encoding:

0	0	1	1	DS	1	0	0		s	s	s	s	1	d	d	d

byte 3: offset8

SUBB Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16) - (C)
Encoding:

0	0	1	1	DS	1	0	1		d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUBB [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs) - (C)
Encoding:

0	0	1	1	DS	1	0	1

s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUBB Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	1	1	DS	0	1	1

d	d	d	d	0	s	s	s

SUBB [Rd+], Rs

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	1	1	DS	0	1	1

s	s	s	s	1	d	d	d

SUBB direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) <-- (direct) - (Rs) - (C)
Encoding:

0	0	1	1	DS	1	1	0

s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

SUBB Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (Rd) - (direct) - (C)
Encoding:

0	0	1	1	DS	1	1	0		d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

SUBB Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- (Rd) - #data8 - (C)
Encoding:

1	0	0	1	0	0	0	1		d	d	d	d	0	0	1	1

byte 3: #data8

SUBB Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rd) - #data16 - (C)
Encoding:

1	0	0	1	1	0	0	1		d	d	d	d	0	0	1	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
Encoding:

1	0	0	1	0	0	1	0		0	d	d	d	0	0	1	1

byte 3: #data8

SUBB [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
Encoding:

1	0	0	1	1	0	1	0

0	d	d	d	0	0	1	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
(Rd) <-- (Rd) + 1
Encoding:

1	0	0	1	0	0	1	1

0	d	d	d	0	0	1	1

byte 3: #data8

SUBB [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1

0	d	d	d	0	0	1	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8 - (C)
Encoding:

1	0	0	1	0	1	0	0

0	d	d	d	0	0	1	1

byte 3: offset8
byte 4: #data8

SUBB [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16 - (C)
Encoding:

1	0	0	1	1	1	0	0
0	d	d	d	0	0	1	1

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUBB [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8 - (C)
Encoding:

1	0	0	1	0	1	0	1
0	d	d	d	0	0	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUBB [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16 - (C)
Encoding:

1	0	0	1	1	1	0	1
0	d	d	d	0	0	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUBB direct, #data8

Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct) - #data8 - (C)
Encoding:

1	0	0	1	0	1	1	0
0	direct: 3 bits			0	0	1	1

byte 3: lower 8 bits of direct
byte 4: #data8

SUBB direct, #data16

Bytes: 5
Cycles: 4
Operation: (direct) <-- (direct) - #data16 - (C)
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	0	1	1

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

TRAP **Software Trap**

Syntax: TRAP #data4

Operation:
(PC) <-- (PC) + 2
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (trap vector (#data4))
(PC.15-0) <-- code memory (trap vector (#data4))
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes the specified software trap. The invoked routine is determined by branching to the specified vector table entry point. The RETI, return from interrupt, instruction is used to resume execution after the trap routine has been completed. A trap acts like an immediate interrupt, using a vector to call one of several pieces of code that will be executed in system mode. This may be used to obtain system services for application code, such as altering the data segment register. This is described in more detail in the section on interrupts and exceptions.

Note: The address of the exception handling routine must be word aligned as the PC is forced to an even address before vectoring to the service routine.

Size: None

Flags Updated: none

Bytes: 2
Cycles: 23/19 (PZ)

Encoding:

1	1	0	1	0	1	1	0

0	0	1	1	#data4

XCH Exchange

Syntax: XCH dest, src

Operation: dest <--> src

Description: The data specified by the source and destination operands is exchanged.

Size: Byte-Byte, word-word.

Flags Updated: none

XCH Rd, Rs

Bytes: 2
Cycles: 5
Operation: (Rd) <--> (Rs)
Encoding:

0	1	1	0	DS	0	0	0

d	d	d	d	s	s	s	s

XCH Rd, [Rs]

Bytes: 2
Cycles: 6
Operation: (Rd) <--> ((WS:Rs))
Encoding:

0	1	0	1	DS	0	0	0

d	d	d	d	0	s	s	s

XCH Rd, direct

Bytes: 3
Cycles: 6
Operation: (Rd) <--> (direct)
Encoding:

1	0	1	0	DS	0	0	0

d	d	d	d	1	direct: 3 bits

byte 3: lower 8 bits of direct

XOR Exclusive OR

Syntax: XOR dest, src

Operation: dest <- dest (XOR) src

Description: The byte or word specified by the source operand is bitwise logically XORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

XOR Rd, Rs

Bytes: 2
Cycles: 3
Operation: (Rd) <-- (Rd) (XOR) (Rs)
Encoding:

0	1	1	1	DS	0	0	1

d	d	d	d	s	s	s	s

XOR Rd, [Rs]

Bytes: 2
Cycles: 4
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
Encoding:

0	1	1	1	DS	0	1	0

d	d	d	d	0	s	s	s

XOR [Rd], Rs

Bytes: 2
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
Encoding:

0	1	1	1	DS	0	1	0

s	s	s	s	1	d	d	d

XOR Rd, [Rs+offset8]

Bytes: 3
Cycles: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset8)
Encoding:

0	1	1	1	DS	1	0	0		d	d	d	d	0	s	s	s

byte 3: offset8

XOR [Rd+offset8], Rs

Bytes: 3
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) (Rs)
Encoding:

0	1	1	1	DS	1	0	0		s	s	s	s	1	d	d	d

byte 3: offset8

XOR Rd, [Rs+offset16]

Bytes: 4
Cycles: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset16)
Encoding:

0	1	1	1	DS	1	0	1		d	d	d	d	0	s	s	s

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

XOR [Rd+offset16], Rs

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) (Rs)
Encoding:

0	1	1	1	DS	1	0	1		s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

XOR Rd, [Rs+]

Bytes: 2
Cycles: 5
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	1	1	1	DS	0	1	1

d	d	d	d	0	s	s	s

XOR [Rd+], Rs

Bytes: 2
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	1	1	1	DS	0	1	1

s	s	s	s	1	d	d	d

XOR direct, Rs

Bytes: 3
Cycles: 4
Operation: (direct) <-- (direct) (XOR) (Rs)
Encoding:

0	1	1	1	DS	1	1	0

s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

XOR Rd, direct

Bytes: 3
Cycles: 4
Operation: (Rd) <-- (Rd) (XOR) (direct)
Encoding:

0	1	1	1	DS	1	1	0

d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct

XOR Rd, #data8

Bytes: 3
Cycles: 3
Operation: (Rd) <-- (Rd) (XOR) #data8
Encoding:

1	0	0	1	0	0	0	1		d	d	d	d	0	1	1	1

byte 3: #data8

XOR Rd, #data16

Bytes: 4
Cycles: 3
Operation: (Rd) <-- (Rd) (XOR) #data16
Encoding:

1	0	0	1	1	0	0	1		d	d	d	d	0	1	1	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd], #data8

Bytes: 3
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
Encoding:

1	0	0	1	0	0	1	0		0	d	d	d	0	1	1	1

byte 3: #data8

XOR [Rd], #data16

Bytes: 4
Cycles: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
Encoding:

1	0	0	1	1	0	1	0		0	d	d	d	0	1	1	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+], #data8

Bytes: 3
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
(Rd) <-- (Rd) + 1

Encoding:

1	0	0	1	0	0	1	1		0	d	d	d	0	1	1	1

byte 3: #data8

XOR [Rd+], #data16

Bytes: 4
Cycles: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
(Rd) <-- (Rd) + 2
Encoding:

1	0	0	1	1	0	1	1		0	d	d	d	0	1	1	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+offset8], #data8

Bytes: 4
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data8
Encoding:

1	0	0	1	0	1	0	0		0	d	d	d	0	1	1	1

byte 3: offset8
byte 4: #data8

XOR [Rd+offset8], #data16

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data16
Encoding:

1	0	0	1	1	1	0	0		0	d	d	d	0	1	1	1

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

XOR [Rd+offset16], #data8

Bytes: 5
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data8
Encoding:

1	0	0	1	0	1	0	1

0	d	d	d	0	1	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

XOR [Rd+offset16], #data16

Bytes: 6
Cycles: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data16
Encoding:

1	0	0	1	1	1	0	1

0	d	d	d	0	1	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

XOR direct, #data8

Bytes: 4
Cycles: 4
Operation: (direct) <-- (direct) (XOR) #data8
Encoding:

1	0	0	1	0	1	1	0

0	direct: 3 bits	0	1	1	1

byte 3: lower 8 bits of direct
byte 4: #data8

XOR direct, #data16

Bytes: 5
Cycles: 4
Operation: (direct) <-- (direct) (XOR) #data16
Encoding:

1	0	0	1	1	1	1	0

0	direct: 3 bits	0	1	1	1

byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

6.6 Summary Of Illegal Operand Combinations On The XA

All but one case are instructions that specify or imply 2 write operations to a single register file location within a single instruction. The other case is a possible corruption of the source register data by an auto-increment before it is read.

The following instructions, where both operands specify the same register:

Instruction(s) affected	Reason for illegal combination
(any op) Rx, [Rx+]	Auto-increment plus explicit write
mov [Rx+], [Rx+]	Double auto-increment of one register
(any op) [Rx+], Rx	Auto-increment write may corrupt the source register before it is read
NORM Rx, Rx	Result and shift count stored in the same register
XCH Rx, Rx	Double write of a single register

Instruction(s) affected	Reason for illegal combination
(any op) [Rx+], Ry	Auto-increment plus indirect write to same register
(any op) [Rx+], [Ry+]	Auto-increment plus indirect write to same register
(any op) [Rx+], #data	Auto-increment plus indirect write to same register
XCH Rx, [Rx]	Indirect write plus explicit write to the same register
XCH Rx, direct	Direct write plus explicit write to the same register

Special cases:

Instruction(s) affected	Reason for illegal combination
POP R7	Stack pointer auto-increment plus explicit write to R7/SP

7 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices that are to be accessed, then generates a strobe for the required operation, with data passing in or out on the data bus. Typical bus operations are code read, data read, and data write. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

The following discussion is based on the standard version of the XA external bus. Some specific XA derivatives may have a different implementation of the external bus, or no external bus at all.

7.1 External Bus Signals

For flexibility, the standard XA external bus supports 8 or 16-bit data transfers and a user selectable number of address bits. The maximum number of address lines varies by derivative but may be up to 24. A standard set of bus control signals coordinates activity on the bus. These are described in the following sections.

7.1.1 $\overline{\text{PSEN}}$ - Program Store Enable

The program store enable signal is used to activate an external code memory, such as an EPROM. This active low signal is typically connected to the Output Enable ($\overline{\text{OE}}$) pin of an external EPROM. $\overline{\text{PSEN}}$ remains high when a code read is not in progress.

7.1.2 $\overline{\text{RD}}$ - Read

The bus read signal is also active low. Activity of this signal indicates data read operations on the external bus. $\overline{\text{RD}}$ is typically connected to the pin of the same name on an external peripheral device.

7.1.3 $\overline{\text{WRL}}$ - Write Low Byte

$\overline{\text{WRL}}$ is the external bus data write strobe. It is typically connected to the $\overline{\text{WR}}$ pin of an external peripheral device. When the XA external bus is used in the 16-bit mode, this strobe applies only to the lower data byte, allowing byte writes on the 16-bit bus. The $\overline{\text{WRL}}$ signal is active low.

7.1.4 $\overline{\text{WRH}}$ - Write High Byte

For a 16-bit data bus, a signal similar to $\overline{\text{WRL}}$, but for the upper data byte is needed. The active low signal $\overline{\text{WRH}}$ serves this purpose.

7.1.5 ALE - Address Latch Enable

Since a portion of the XA external bus is used for multiplexed address and data information, that part of the address must be latched outside of the XA so that it will remain constant during the subsequent read or write operation. The active high ALE signal directs the external latch to allow information to be stored for a data address or a code address. The external latch must close and retain this address when the ALE signal ends, by going low (inactive).

7.1.6 Address Lines

Some of the address lines used by the external bus interface are driven during a complete bus operation and do not need to be latched. In the standard XA bus interface, the lower four address lines are always driven and unlatched in this manner. This is done specifically as part of the optimization of the bus for fetching instructions from external code memory at high speed. This feature will be explained in detail in a later section.

7.1.7 Multiplexed Address and Data Lines

The part of the bus that is used for data transfer is also used for address output from the XA. Prior to asserting the strobe for the bus operation about to be performed, the XA outputs the address for the operation. On the multiplexed portion of the bus, this address is captured by an external latch, as commanded by the ALE signal. After that is done, this part of the bus is free to be used for data transfer either into or out of the XA. The control signals $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, and $\overline{\text{WRH}}$ determine what type of bus operation takes place.

7.1.8 WAIT - Wait

The WAIT input allows wait states to be inserted into any external bus operation. If WAIT is asserted (high) after a bus control strobe ($\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, or $\overline{\text{WRH}}$) is driven by the XA, that bus operation is stretched, and that control strobe continues to be driven by the XA until WAIT goes low again. For this feature to be used, an external circuit must be present to generate the WAIT signal at the appropriate times.

The XA has an internal bus configuration feature that allows programming the various types of external bus cycles to different lengths, so that in most applications the WAIT line will not be needed. This feature will be explained in detail in a later section.

7.1.9 $\overline{\text{EA}}$ - External Access

The $\overline{\text{EA}}$ input determines whether the XA operates in single-chip mode, or begins running code from the internal program memory after reset. If $\overline{\text{EA}}$ is low as Reset goes high, the first code fetch (and all others after that) is made off-chip. If $\overline{\text{EA}}$ is high as Reset goes high, the XA will execute the on-chip code first, but will still attempt to execute instructions from external memory at addresses above the limit of on-chip code. The level on the $\overline{\text{EA}}$ pin is latched as reset goes high, so whatever mode is selected remains valid until the next reset.

On some XA derivatives, the pin used for the $\overline{\text{EA}}$ function may be shared with another function that becomes active after the XA begins code execution.

7.1.10 BUSW - Bus Width

The external XA bus may be configured to be 8 or 16 bits in width. The XA allows the bus width to be programmed in 2 ways. In a system where instructions are initially fetched from on-chip code memory, the user program can configure the external bus size (and many other aspects of the bus) prior to the bus actually being used.

When the initial code fetches must be done using off-chip code memory, however, the XA must know the bus width before the first off-chip code fetch can begin.

On some XA derivatives, the BUSW function may share a pin with some other function. In this case, the level on the BUSW pin is latched as Reset is released and that selection is kept until the next Reset. The secondary function on that pin will be active after Reset when the processor begins executing code normally.

Unlike the $\overline{\text{EA}}$ function, the bus width set by the BUSW pin at reset may be over-ridden by a user program, making setting by use of the BUSW pin unnecessary in most systems. Settings in the Bus Configuration Register allow changing the bus size under program control. This feature is covered in more detail in the next section.

7.2 Bus Configuration

The standard XA external bus has a number of configuration options. In addition to the data bus width selection discussed previously, the number of address lines used for external accesses is programmable, as is the bus timing.

7.2.1 8-Bit and 16-Bit Data Bus Widths

The standard XA external bus allows both 8-bit and 16-bit bus widths. The bus width is determined by the value of the BUSW pin as Reset is released, unless a user program overrides that setting by writing to the Bus Configuration Register (BCR), shown in Figure7.1.

BCR	-	-	-	WAITD	BUSD	BC2	BC1	BC0

WAITD:	WAIT disable. Causes the XA external bus interface to ignore the value on the WAIT input. This allows tying the WAIT input high for applications that use internal code and do not need the WAIT function.
BUSD:	Bus disable. Causes XA external bus functions to be disabled permanently. The primary purpose of this is to allow prevention of inadvertent activation of the bus by an instruction pre-fetch when the XA is executing code near the end of the on-chip code memory.
BC2 - BC0:	These bits select the XA external bus configuration, specifically the number of data bits and the number of address lines. The supported combinations are shown below. 000 : 8-bit data bus, 12 address lines 001 : 8-bit data bus, 16 address lines 010 : 8-bit data bus, 20 address lines 011 : 8-bit data bus, 24 address lines 100 : < function reserved > 101 : < function reserved > 110 : 16-bit data bus, 20 address lines 111 : 16-bit data bus, 24 address lines
"-"	Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Figure 7.1 Bus Configuration Register (BCR)

Figures 7.2 and 7.3 show the address and data functions present on XA bus related pins when used with each available bus width.

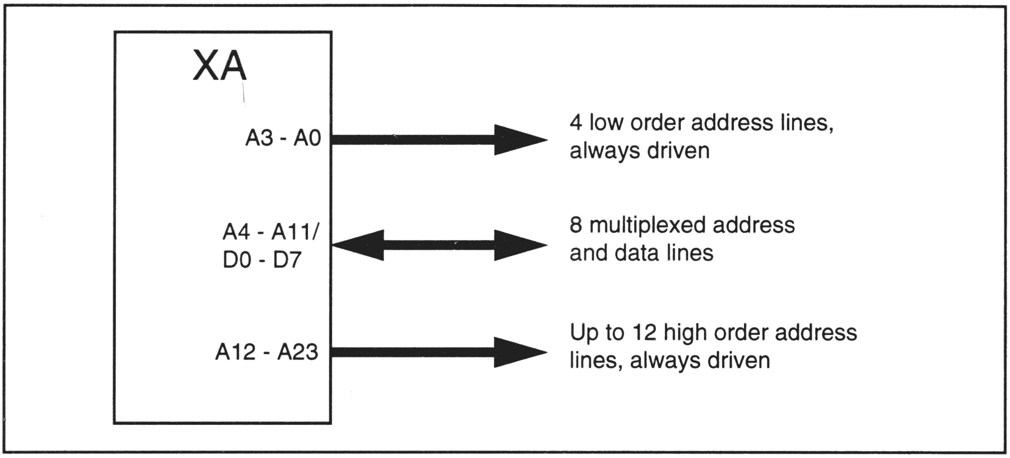

XA
A3 - A0
4 low order address lines, always driven
A4 - A11/ D0 - D7
8 multiplexed address and data lines
A12 - A23
Up to 12 high order address lines, always driven

Figure 7.2 8-Bit External Bus Configuration

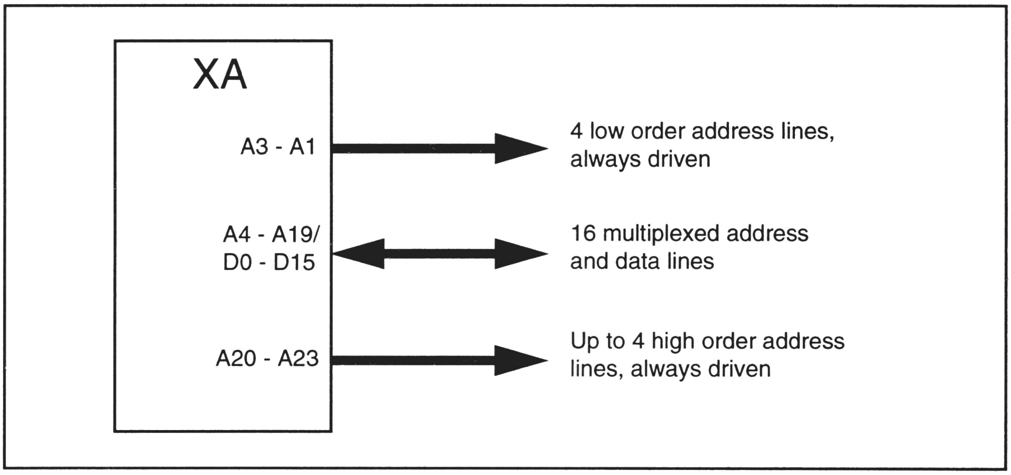

XA
A3 - A1
4 low order address lines, always driven
A4 - A19/ D0 - D15
16 multiplexed address and data lines
A20 - A23
Up to 4 high order address lines, always driven

Figure 7.3 16-Bit External Bus Configuration

7.2.2 Typical External Device Connections

Many possibilities exist for connecting and using external devices with the XA bus. The bus will support EPROMs, RAMs, and other memory devices, as well as peripheral devices such as UARTs, and parallel port expanders. The following diagrams show a generalized connection of devices for 8-bit and 16-bit XA bus modes.

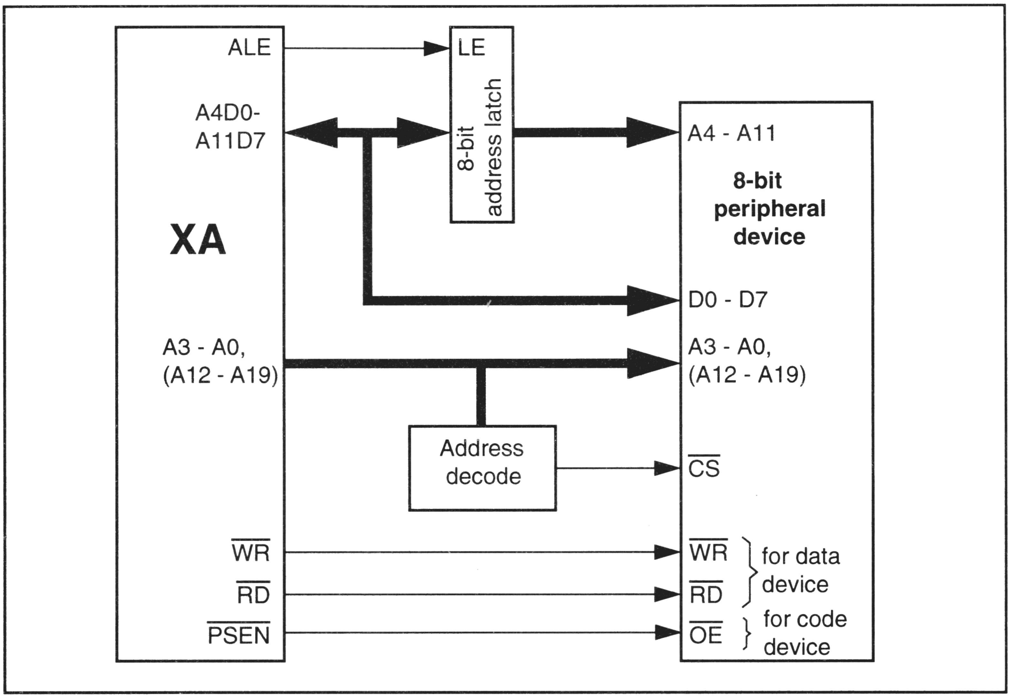

ALE
LE
A4D0-
A11D7
8-bit
address latch
A4 - A11
8-bit
peripheral
device
XA
D0 - D7
A3 - A0,
(A12 - A19)
A3 - A0,
(A12 - A19)
Address
decode
CS
WR
WR
for data
device
RD
RD
PSEN
OE
for code
device

Figure 7.4 Typical XA External Bus Connections for 8-Bit Peripheral Devices

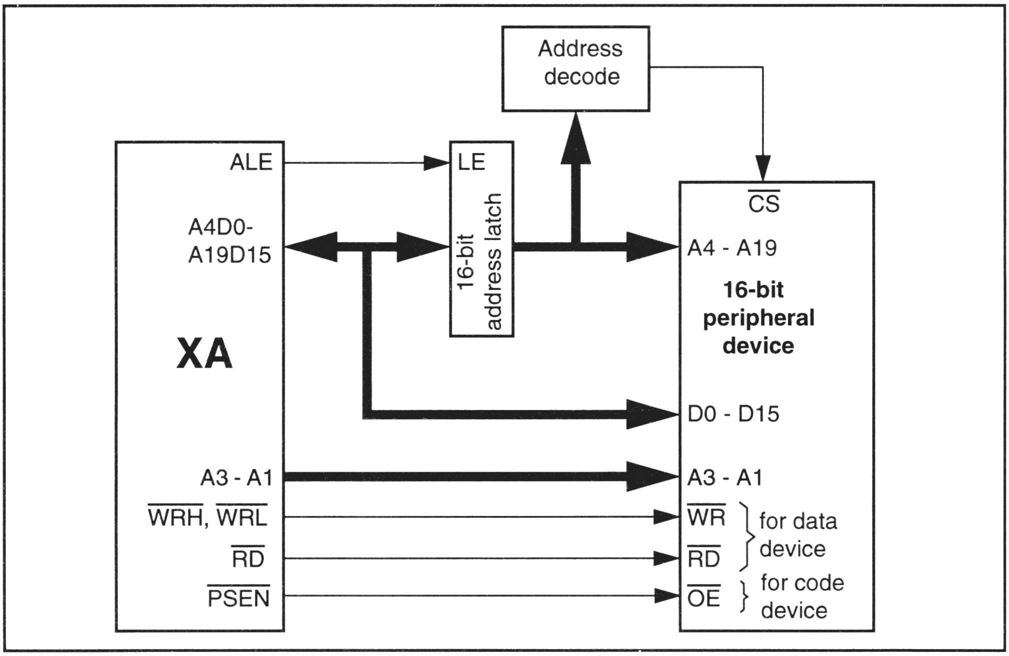

Address
decode
ALE
LE
CS
A4D0-
A19D15
16-bit
address latch
A4 - A19
16-bit
peripheral
device
XA
D0 - D15
A3 - A1
A3 - A1
WRH, WRL
WR
for data
device
RD
RD
PSEN
OE
for code
device

Figure 7.5 Typical XA External Bus Connections for 16-Bit Peripheral Devices

7.3 Bus Timing and Sequences

The standard XA external bus allows programming the widths of the bus control signals ALE, PSEN, WRL, WRH, and RD. There is also an option to extend the data hold time after a write operation. The combinations available will allow interfacing most devices to the XA directly without the need for special buffers or a WAIT state generator.

7.3.1 Code Memory

Interfacing with external code memory, typically in the form of EPROMs, is enabled by the $\overline{\text{PSEN}}$ control signal. If the XA is configured to execute internal code memory at reset, by the setting of the $\overline{\text{EA}}$ pin, it will automatically begin to fetch external code if the program crosses the boundary from internal to external code space. The location of this boundary varies for different XA derivatives, depending on the size of the internal code memory for each part.

Since the XA employs a pre-fetch queue in order to optimize instruction execution times, fetching of external instructions may begin before program execution actually crosses the on/off-chip code memory boundary. If a branch or subroutine return is located near the end of on-chip code memory, the off-chip fetch would be unnecessary, and may in fact cause problems if the XA ports that implement the external bus are being used for other purposes. For this reason, the BUSD (bus disable) bit in the Bus Configuration Register (BCR) is provided to prevent the XA from using the external bus for code or data operations.

Code Read with ALE

The classic operation of a multiplexed address and data bus involves the issuance of an address, along with its associated control signal, for every bus cycle. The XA uses the bus control signal ALE to indicate that an address is on the bus that must be latched through the following code or data operation. The following diagram shows a code memory fetch in a cycle using ALE.

Burst Code Read (No ALE)

The XA does not always require an ALE cycle for every code fetch. This feature is included specifically to improve performance when the XA executes code from external memory, while increasing the access time available for the external memory device. Because the lower four address lines of the external bus are always driven, not multiplexed, the XA can access up to 16 bytes (or 8 words) of sequential code memory each time an ALE is issued. This type of fast sequential code read is called a burst read. Of course, any type of jump, branch, interrupt, or other change in sequential program flow will require an ALE in order to change the code fetch address in a non-sequential manner. Any data operation (read or write) on the XA external bus also requires an ALE cycle and will cause any subsequent external code fetch to begin with an ALE cycle also.

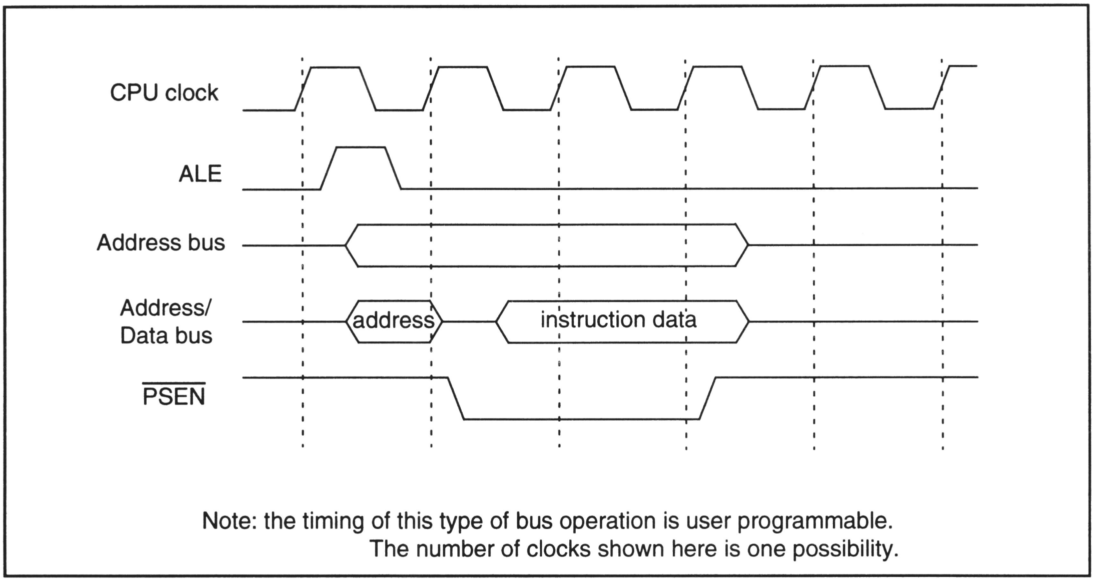

CPU clock
ALE
Address bus
Address/
Data bus
PSEN
address
instruction data
Note: the timing of this type of bus operation is user programmable.
The number of clocks shown here is one possibility.

Figure 7.6 Typical External Code Read Using ALE

The following diagram shows a typical sequential code fetch where no ALE is issued between code reads. Also note that the $\overline{\text{PSEN}}$ bus control signal does not toggle, but remains asserted throughout the burst code read

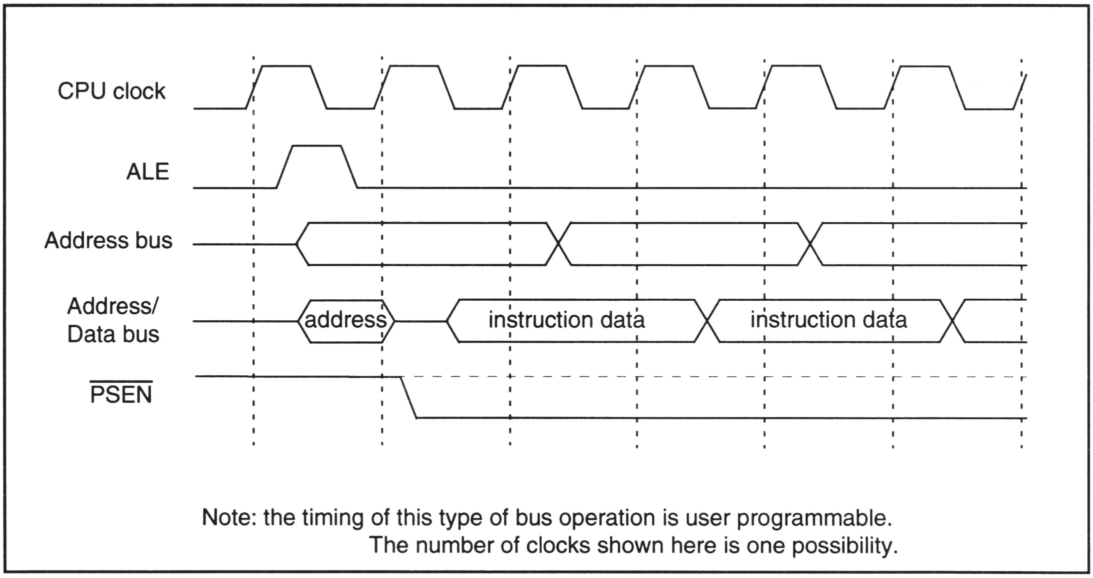

CPU clock
ALE
Address bus
Address/
Data bus
PSEN
address
instruction data
instruction data
Note: the timing of this type of bus operation is user programmable.
The number of clocks shown here is one possibility.

Figure 7.7 Burst Mode (Sequential) External Code Read

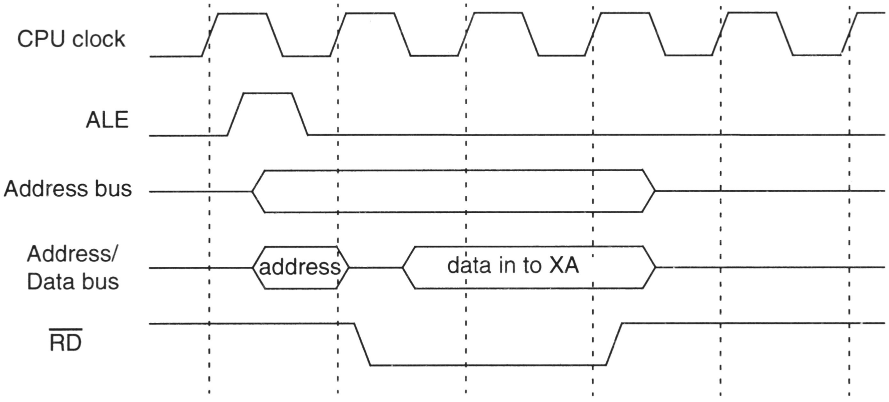
7.3.2 Data Memory

Reads and writes on the XA external bus are controlled through the use of the $\overline{\text{RD}}$, $\overline{\text{WRL}}$, and $\overline{\text{WRH}}$ signals. Since the XA bus supports both 8-bit and 16-bit widths, as well as byte and word read and write operations, several different versions of the basic bus cycles are possible. These are described in the following sections.

Data memory, like code memory, has a boundary where the internal data memory ends, and above which the XA will switch to the external bus in order to act on data memory. This on/off-chip data memory boundary may be in a different place for various XA derivatives, depending upon the amount of internal data memory built into a specific derivative.

Typical Data Read

A simple byte read on an 8-bit bus or any read on a 16-bit bus both begin with an ALE cycle, where the XA presents the address of the data location that is to be read on the bus. This is followed by the assertion of the $\overline{\text{RD}}$ strobe, that causes the external device to present its data on the bus. This process is shown in the diagram below.



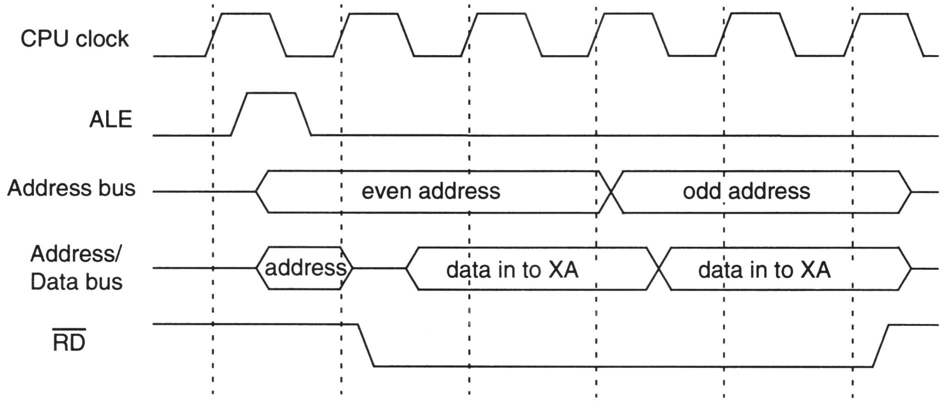
Note: the timing of this type of bus operation is user programmable. The number of clocks shown here is one possibility.

Figure 7.8 Typical External Data Read

Word Read on an 8-Bit Data Bus

When the XA external bus is configured for an 8-bit data width, a word read operation is automatically performed as two byte reads at sequential addresses. Since the XA CPU requires word operations to be performed at even addresses, the second half of any word read on a byte-wide bus always uses the same upper address latched by ALE. for this operation, the low order byte first is read at the even byte address, then the high order byte is read at the next (odd) address. So, only one ALE is required in this case. The diagram below shows this sequence.

Figure 7.9 Word Read on 8-Bit Data Bus



Note: the timing of this type of bus operation is user programmable.

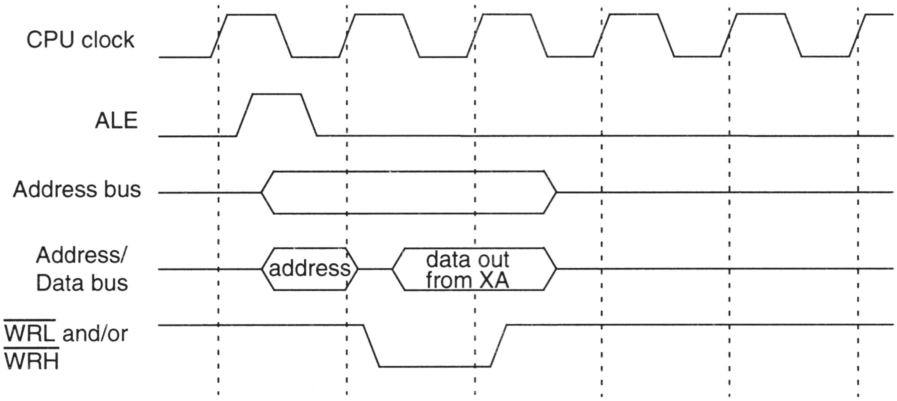
Byte Read on a 16-Bit Data Bus

When an instruction causes a read of one byte of data from the external bus, when it is configured for 16-bit width, a simple read operation is performed. This results in 16 bits of data being received by the XA, which uses only the byte that was requested by the program. There is no way to distinguish a byte read from a word read on the external bus when it is configured for a 16-bit width.

Typical Data Write

A data write operation begins with an ALE cycle, like a read operation, followed by the assertion of one or both of the write strobes, $\overline{\text{WRL}}$ and $\overline{\text{WRH}}$. This simple bus cycle applies to byte writes on an 8-bit data bus and all writes on a 16-bit data bus.

A byte write on an 8-bit data bus will always use only the $\overline{\text{WRL}}$ strobe. A byte write on a 16-bit data bus will always use either the $\overline{\text{WRL}}$ or $\overline{\text{WRH}}$ strobe, depending on whether the byte is at an even or odd address. A word write on a 16-bit bus requires the assertion of both the $\overline{\text{WRL}}$ and $\overline{\text{WRH}}$ strobes. The simple data write cycle is shown below.



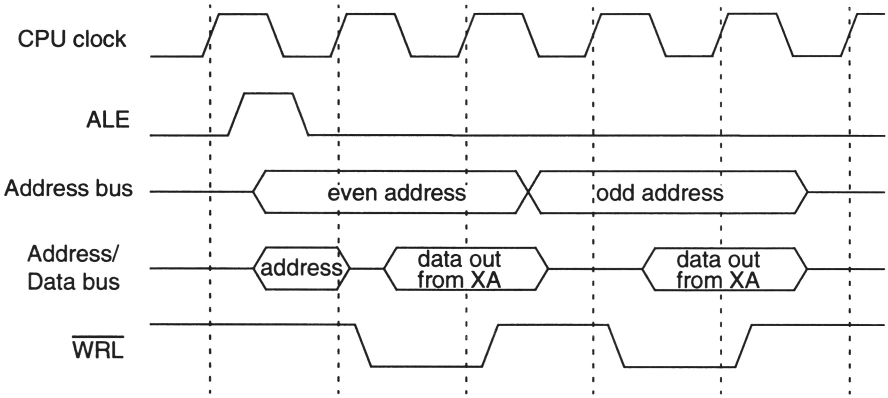
CPU clock
ALE
Address bus
Address/
Data bus
address
data out
from XA
$\overline{\text{WRL}}$ and/or
$\overline{\text{WRH}}$

Note: the timing of this type of bus operation is user programmable.
The number of clocks shown here is one possibility.

Figure 7.10 Typical External Data Write

Word Write on an 8-Bit Data Bus

When a word write operation is done with the bus configured to an 8-bit width, the XA automatically performs two byte writes. First, the low order byte is written (at the even byte address), then the high order byte is written at the next (odd) address. As with a word read on an 8-bit bus, this requires only a single ALE cycle at the beginning of the process. This sequence is shown in the following diagram.


CPU clock
ALE
Address bus
even address
odd address
Address/
Data bus
address
data out
from XA
data out
from XA
$\overline{WRL}$

Note: the timing of this type of bus operation is user programmable.
The number of clocks shown here is one possibility.

Figure 7.11 Word Write on 8-Bit Data Bus

External Bus Signal Timing Configuration

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, data read and write cycle lengths, and data hold time. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

Programmable bus timing is controlled by settings found in the Bus Timing Register SFRs, named BTRH, and BTRL, shown in figures 7.12 and 7.13.

BTRH	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0

DW1, DW0: Data Write without ALE. Applies only to the second half of a 16-bit write operation when the bus is configured to 8 bits.
00 : Data write cycle is 1 clock in duration.
01 : Data write cycle is 2 clocks in duration.
10 : Data write cycle is 3 clocks in duration.
11 : Data write cycle is 4 clocks in duration.

DWA1, DWA0: Data Write with ALE. Selects the length (in CPU clocks) of the entire data write cycle, including ALE.
00 : Data write cycle is 2 clocks in duration.
01 : Data write cycle is 3 clocks in duration.
10 : Data write cycle is 4 clocks in duration.
11 : Data write cycle is 5 clocks in duration.

DR1, DR0: Data Read without ALE. Applies only to the second half of a 16-bit read operation when the bus is configured to 8 bits.
00 : Data read cycle is 1 clock in duration.
01 : Data read cycle is 2 clocks in duration.
10 : Data read cycle is 3 clocks in duration.
11 : Data read cycle is 4 clocks in duration.

DRA1, DRA0: Data Read with ALE. Selects the length (in CPU clocks) of the entire data read cycle, including ALE.
00 : Data read cycle is 2 clocks in duration.
01 : Data read cycle is 3 clocks in duration.
10 : Data read cycle is 4 clocks in duration.
11 : Data read cycle is 5 clocks in duration.

Note: see text regarding disallowed bus timing combinations.

Figure 7.12 Bus Timing Register High Byte (BTRH)

BTRL	WM1	WM0	ALEW	-	CR1	CR0	CRA1	CRA0

WM1:	Write Mode 1. Selects the width of the write pulse. 0 : Write pulse (WR) width is 1 CPU clock. 1 : Write pulse (WR) width is 2 CPU clocks.
WM0:	Write Mode 0. Selects the data hold time. 0 : Data hold time is minimum. 1 : Data hold time is 1 CPU clock.
ALEW:	ALE width selection. Determines the duration of ALE pulses. 0 : ALE width is one half of one CPU clock. 1 : ALE width is one and a half CPU clocks.
CR1, CR0:	Code Read. Selects the length of a code read cycle when ALE is not used. 00 : Code read cycle is 1 clocks in duration. 01 : Code read cycle is 2 clocks in duration. 10 : Code read cycle is 3 clocks in duration. 11 : Code read cycle is 4 clocks in duration.
CRA1, CRA0:	Code Read with ALE. Selects the length of a code read cycle when ALE is used prior to $\overline{\text{PSEN}}$ being asserted. 00 : Code read cycle is 2 clocks in duration. 01 : Code read cycle is 3 clocks in duration. 10 : Code read cycle is 4 clocks in duration. 11 : Code read cycle is 5 clocks in duration.
"-"	Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Note: see text regarding disallowed bus timing combinations.

Figure 7.13 Bus Timing Register Low Byte (BTRL)

Disallowed Bus Timing Configurations

Some possible combinations of bus timing register settings do not make sense and the XA cannot produce working bus signals that match those settings. The disallowed combinations occur where the sum of the specified components of a bus cycle exceed the specified length of the entire cycle. Four simple rules define the allowed/disallowed combinations. Violating these rules may result in incomplete bus cycles, for example a data read cycle in which an address and ALE pulse are output, but no read strobe ($\overline{\text{RD}}$) is produced.

For data write cycles on the external bus there are two conditions that must be met. The first applies to data write cycles with no ALE:

$$\text{WM1} + \text{WM0} \leq \text{DW1:0}$$

This says that the sum of the values of the WM1 and WM0 fields must be less than or equal to the value of the DW field. Note that this is the value of the fields, not the timing durations that they specify. The other rules use the same structure, as follows.

A second requirement applies to write cycles with ALE:

$$\text{ALEW} + \text{WM1} + \text{WM0} \leq \text{DWA1:0}$$

The configuration for Data Read timing has only one requirement:

$$\text{ALEW} \leq \text{DRA} + 1$$

The configuration for Code Read timing also has only one requirement:

$$\text{ALEW} \leq \text{CRA} + 1$$

7.3.3 Reset Configuration

Upon reset, at the time of power up or later, the XA bus is initially configured in certain ways. As previously discussed, the pins $\overline{\text{EA}}$ and BUSW select whether the XA will begin operation from internal code, and whether the bus will be 8-bits or 16-bits.

The values for the programmable bus timing are also set to a default value at reset. All of the timing values are set to their maximum, providing the slowest bus cycles. This setting allows for the slowest external devices that may be sued with the XA without WAIT generation logic. The user program should set the bus timing to the correct values for the specific application in the system initialization code. Refer to the data sheet for a particular XA derivative for details of the values found in registers and SFRs after reset.

7.4 Ports

I/O ports on any microcontroller provide a connection to the outside world. The capabilities of those I/O ports determine how easily the microcontroller can be interfaced to the various external devices that make up a complete application. The standard XA I/O ports provide a high degree of versatility through the use of programmable output modes and allow easy connection to a wide variety of hardware.

7.4.1 I/O Port Access

The standard on-chip I/O ports of the XA are accessed as SFRs. The SFR names used for these ports begin with port 0, called P0. Port numbers and names go up in sequence from there, to the number of ports on a specific XA derivative. Ports are normally identified by their names in assembler source code, such as: "MOV P1,#0". This instruction causes the value 0 to be written to port 1.

XA I/O ports are typically bit addressable, meaning that individual port bits are readable, writable, and testable. An instruction using a port bit looks like this: "SETB P2.1". This particular example would result in the second lowest bit in port 2 (bit 1) having a 1 written to it.

Reading of a Port Pin Versus the Port Latch

Each I/O port has two important logic values associated with it. The first is the contents of the port latch. When data is written to a port, it is stored in the port latch. The second value is the logic level of the actual port pin, which may be different than the port latch value, especially if a port pin is being used as an input.

When a port is explicitly read by an instruction, the value returned is that from the pin. When a port is read intrinsically, in order to perform some operation and store the value back to the port, the port latch is read. This type of operation is called a read-modify-write.

1) The following instructions cause read-modify-write operations, and read the port latch when a port or port bit is specified as the destination:

```
ADD     Px, ...
ADDC    Px, ...
ADDS    Px, ...
AND     Px, ...
DJNZ    Px, ...
OR      Px, ...
SUB     Px, ...
SUBB    Px, ...
XOR     Px, ...

CLR     Px.y
JBC     Px.y, rel8
MOV     Px.y, C
SETB    Px.y
```

2) The following instruction reads the port pins when a port is specified as the destination operand:

```
CMP     Px, ...
```

3) When a port or port bit is specified as a source in any instruction, the port pin is always read.

Figure 7.14 How ports are read.

7.4.2 Port Output Configurations

Standard XA I/O ports provide several different output configurations. One is the 80C51 type quasi-bidirectional port output. Others are open drain, push-pull, and high impedance (input only). It is important to note that the port configuration applies to a pin even if that pin is part of the external bus. Bus pins should normally be configured to push-pull mode. Also, the port latches for pins that are to be used as part of the external bus must be set to one (which is the reset state). A zero in a port latch will override bus operations and force a zero on the corresponding bus position.

The port configuration is controlled by settings in two SFRs for each port. One bit in each port configuration register is associated with a port pin in the corresponding bit position. These port configuration SFRs are called: PnCFGA and PnCFGB, where "n" is the port number. So, the configuration registers for port 1 are named P1CFGA and P1CFGB. The table below shows the port control bit combinations and the associated port output modes.

Table 7.1

PnCFGB	PnCFGA	Port Output Mode
0	0	Open drain.
0	1	Quasi-bidirectional (default).
1	0	High impedance.
1	1	Push-pull.

7.4.3 Quasi-Bidirectional Output

The default port output configuration for standard XA I/O ports is the quasi-bidirectional output that is common on the 80C51 and most of its derivatives. This output type can be used as both an input and output without the need to reconfigure the port. This is possible because when the port outputs a logic high, it is weakly driven, allowing an external device to pull the pin low. When the pin is pulled low, it is driven strongly and able to sink a fairly large current. These features are somewhat similar to an open drain output except that there are three pullup transistors in the quasi-bidirectional output that serve different purposes.

One of these pullups, called the "very weak" pullup, is turned on whenever the port latch for a particular pin contains a logic 1. The very weak pullup sources a very small current that will pull the pin high if it is left floating.

A second pullup, called the "weak" pullup, is turned on when the port latch for its associated pin contains a logic 1 and the pin itself is a logic 1. This pullup provides the primary source current for a pin that is outputting a 1, and can drive several TTL loads. If a pin that has a logic 1 on it is pulled low by an external device, the weak pullup turns off, and only the very weak pullup remains on. In order to pull the pin low under these conditions, the external device has to sink enough current to overpower the weak pullup and pull the voltage on the port pin below its input threshold.

The third (and final) pullup is referred to as the "strong" pullup. This pullup is included to speed up low-to-high transitions on a port pin when the port latch changes from 0 to 1. When this occurs, the strong pullup turns on for a brief time, two CPU clocks, pulling the port pin high quickly, then turning off again.

The quasi-bidirectional output structure normally provides a means to have mixed inputs and outputs on port pins without the need for special configurations. However, it has several drawbacks that can be problems in certain situations. For one thing, quasi-bidirectional outputs have a very small source current and are therefore not well suited to driving certain types of loads. They are especially unsuited to directly drive the bases of external NPN transistors, a common method of boosting the current of I/O pins.

Also, since the weak pullup turns off when a port pin is actually low, and the strong pullup turns on only for a brief time, it is possible that under certain port loading conditions, the port pin will get "stuck" low and cannot be driven high. This tends to happen when an external device being driven by the port pin has some leakage to ground that is larger than the current supplied by the very weak pullup of the quasi-bidirectional port output. If there is also a fairly large capacitance on the pin, from a combination of the wiring itself and the pin capacitance of the device(s) connected to the pin, the strong pullup may not succeed in pulling the pin high enough while it is turned on. When the strong pullup is then turned off, the leakage of the external device pulls the pin low again, since only the very weak pullup is turned on at that point and the leakage is greater than the very weak pullup source current. These issues are the reason for enhancing the port configurations of the XA.

A diagram of the quasi-bidirectional output structure is shown in the figure below.

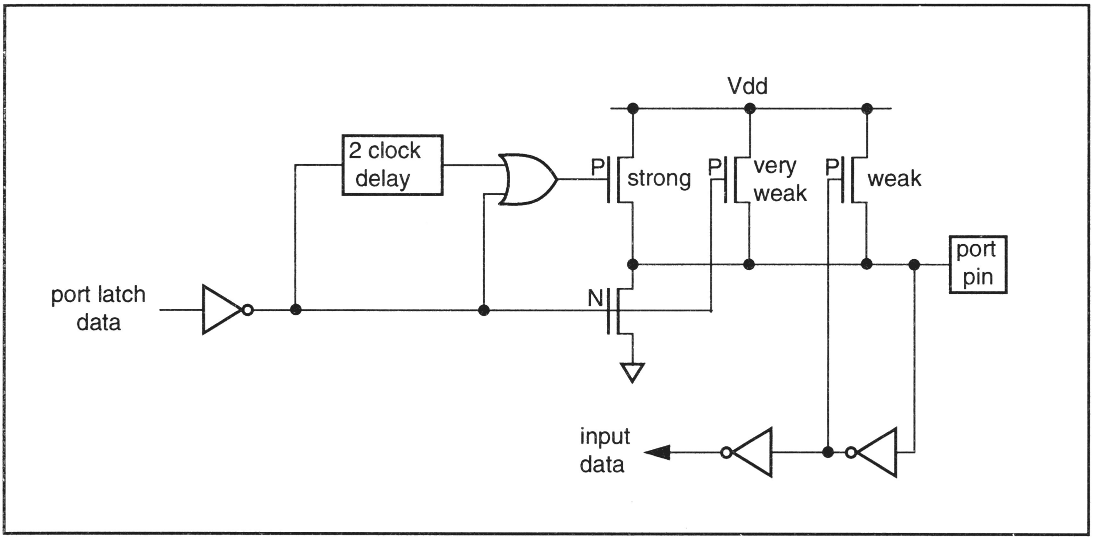


Figure 7.15 Structure of the Quasi-Bidirectional Output Configuration

Open Drain Output

Another port output configuration provided by the standard XA I/O ports is open drain. This configuration turns off all pullups and only drives the pulldown transistor of the port driver when the port latch contains a logic 0. To be used as a logic output, a port configured in this manner must have an external pullup, typically a resistor tied to Vdd. The pulldown for this mode is the same as for the quasi-bidirectional mode.

An advantage of the open drain output is that is may be used to create wired AND logic. Several open drain outputs of various devices can be tied together, and any one of them can drive the wire low, creating a logical AND function without using a logic gate. The figure below show the structure of the open drain output.

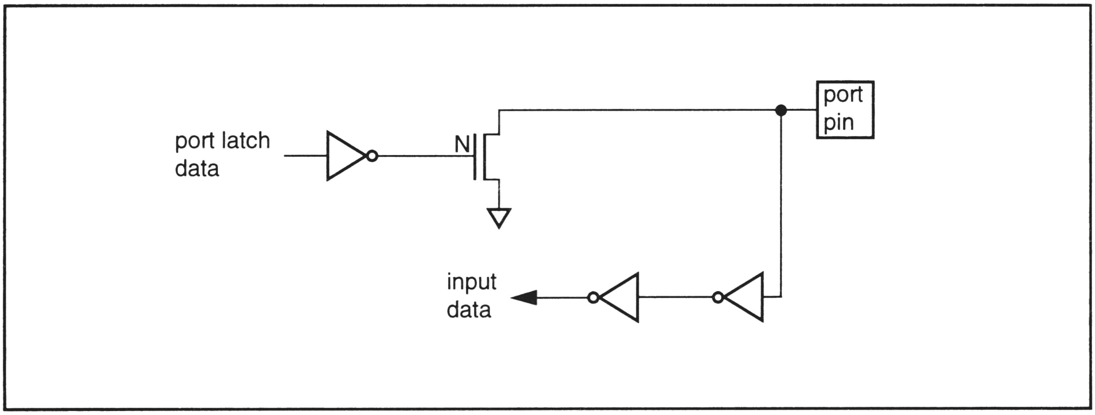


Figure 7.16 Structure of the Open Drain Output Configuration

Push-Pull Output

The push-pull output mode has the same pulldown structure as both the open drain and the quasi-bidirectional output modes, but provides a continuous strong pullup when the port latch contains a logic 1. This mode uses the same pullup as the strong pullup for the quasi-bidirectional mode. The push-pull mode may be used when more source current is needed from a port output. The output structure for this mode is shown below.

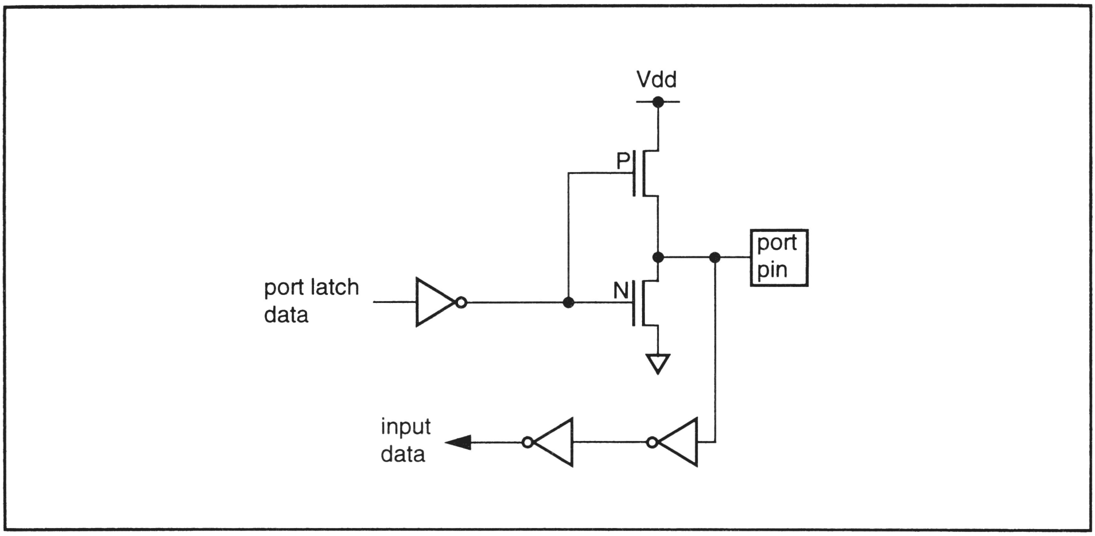


Figure 7.17 Structure of the Push-Pull Output Configuration

High Impedance Output

The final XA port output configuration is called high impedance mode. This mode simply turns all output drivers on a port pin off. Thus, the pin will not source or sink current and may be used effectively as an input-only pin with no internal drivers for an external device to overcome.

7.4.4 Reset State and Initialization

Upon chip reset, all of the port output configurations are set to quasi-bidirectional, and the port latches are written with all ones. The quasi-bidirectional output type is a good default at power-up or reset because it does not source a large amount of current if it is driven by an external device, yet it does not allow the port pin to float. A floating input pin on a CMOS device can cause excess current to flow in the pin's input circuitry, and of course all port pins have input circuits in addition to outputs.

7.4.5 Sharing of I/O Ports with On-Chip Peripherals

Since XA on-chip peripheral devices share device pins with port functions, some care must be taken not to accidentally disable a desired pin function by inadvertently activating another function on the same pin. A peripheral that has an output on a pin will use the I/O port output configuration for that pin (quasi-bidirectional, open drain, push-pull, or high impedance).

The method of sharing multiple functions on a single pin involves a logic AND of all of the functions on a pin. So, if a port latch contains a zero, it will drive that port pin low, and any peripheral output function on that pin is overridden. Conversely, an on-chip peripheral outputting a zero on a pin prevents the contents of the port latch from controlling the output level. It is usually not an issue to avoid turning on an alternate peripheral function on a pin accidentally, since most peripherals must be either explicitly turned on or activated by a write to one of their SFRs. It is more likely that a user program could erroneously write a zero to a port latch bit corresponding to a pin whose with a peripheral function that is being used and therefore disable that function. The simple rule to follow is: never write a zero to a port bit that is associated with an active on-chip peripheral, or that should only be used an input.

8 Special Function Register Bus

The Special Function Register Bus or SFR Bus is the means by which all Special Function Registers are connected to the XA CPU so that they may be read and written by user programs. This includes all of the registers contained in peripherals such as Timers and UARTs, as well as some CPU registers such as the PSW. CPU registers communicate functionally with the CPU via direct connections, but read and write operations performed on them are routed through the SFR bus.

The SFR bus provides a common interface for the addition of any new functions to the XA core, thus supplying the means for building a large and varied microcontroller derivative family. This is illustrated in figure 8.1

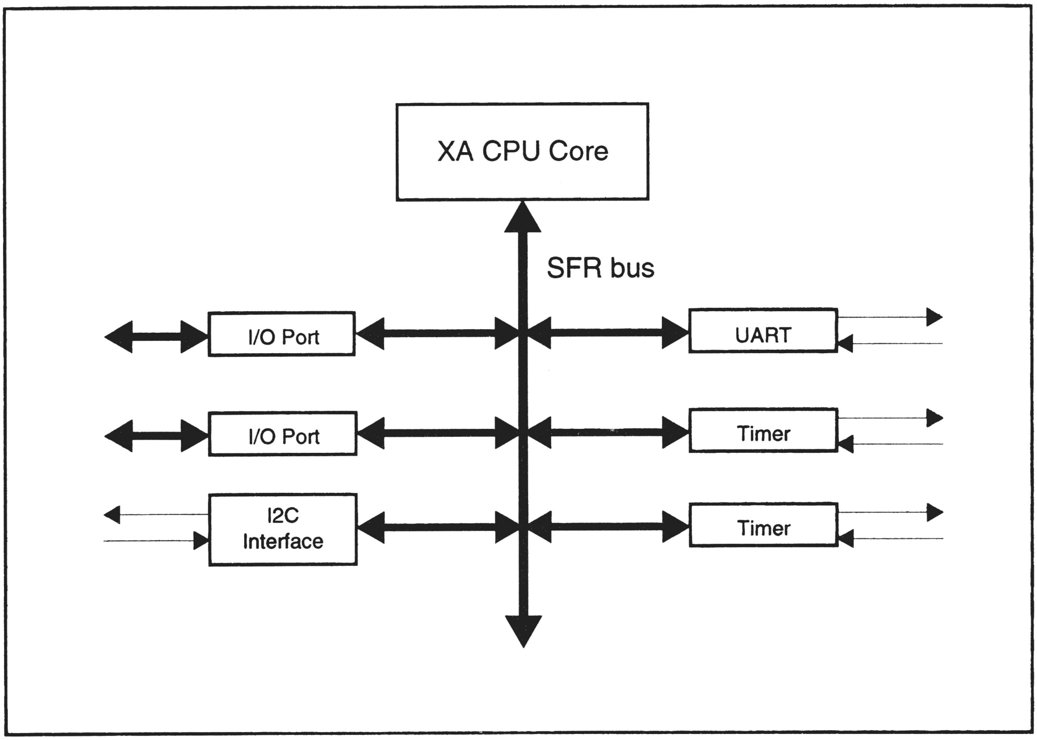


Figure 8.1. Example of peripheral functions connected to the XA SFR bus.

8.1 Implementation and Possible Enhancements

The SFR bus interface is itself not part of the XA CPU core, but a separate functional block. Since the SFR bus controller is a separate block, writes to SFRs may occur simultaneously with the beginning of execution of the next instruction. If the next instruction attempts to access the SFR bus while it is still busy, the instruction execution will stall until the SFR bus becomes available. SFR bus read and write cycles each take 2 CPU clocks to complete. However, the starting time of those 2 clocks has a one clock uncertainty, so the time from the SFR bus controller receiving a request until it is completed can be either 2 or 3 clocks.

The SFR bus implementation on initial XA derivatives is an 8-bit interface. This means that word reads and writes are not allowed. In the future, higher performance XA architecture implementations may expand the capabilities of the SFR bus by supporting 16-bit accesses.

One enhancement to the SFR bus would be to have it divide 16-bit access requests into two 8-bit accesses. This leaves the actual SFR bus width at 8 bits, but allows a user program to act as if it was 16-bits. The highest performance alternative is a full 16-bit SFR bus. This would require extra hardware in the XA to implement, but may eventually become necessary on order to achieve very high performance with some future enhanced XA core implementation.

8.2 Read-Modify-Write Lockout

Some of the SFRs that are accessed via the SFR bus contain interrupt flags and other status bits that are set directly by the peripheral device. When a read-modify-write operation is done on such an SFR, there is a possibility that a peripheral write to a flag bit in the same SFR could occur in the middle of this process. A standard mechanism is defined for the XA to deal with such cases, which is called Read-Modify-Write lockout. A read- modify-write is defined as an operation where a particular SFR is read, altered and written during the execution of a single XA instruction.

The instructions that fit this description are those that write to bits in SFRs and those that modify an entire SFR, except for the MOV instruction. This happens to be the same operations as those that read port latches rather than port pins as specified in Chapter 7, only the SFRs involved are different.

The mechanism used throughout XA peripherals to avoid losing status flags during a read-modify-write operation first involves detecting that such an operation is in progress. A signal from the CPU to the peripherals indicates such a condition. When a peripheral detects this, it prevents the final write operation to just those status flags that it has already updated since the beginning of the instruction. This basically makes it look as if the peripheral flag update happened just after the read-modify-write operation completed, rather than during it. Once the read-modify-write operation is completed, any write to the same SFR by the executing program may affect all bits in the SFR.

9 80C51 Compatibility

Many architectural decisions and features were guided by the goal of 80C51 compatibility when the XA core specification was written. The processor's memory configuration, memory addressing modes, instruction set, and many other things had to be taken into account.

9.1 Compatibility Considerations

Source code compatibility of the XA to the 80C51 was chosen as a goal for many reasons. Complete compatibility with an existing processor is not possible if the new processor is to have substantially higher performance.

The XA architecture makes use of a number of rules for 80C51 compatibility. An 80C51 to XA source code translator program is intended to be the means of providing compatibility between the architectures. For the translator software to be fairly simple, a one-to-one translation for all 80C51 instructions is a major consideration. The XA instruction set includes many instructions that are more powerful than 80C51 instructions and yet perform roughly the same function. 80C51 instruction can therefore be translated into those XA instructions. When this is not the case, an 80C51 instruction may be included in its original form in the XA. The XA memory map and memory addressing modes are also a superset of the 80C51, making source code translation easy to accomplish. Other CPU features are made compatible to the extent that such is possible. In rare cases, when this compatibility could not be provided for some important reason, the changes were kept to the minimum while maintaining the XA goals of high performance and low cost.

9.1.1 Memory Map and Addressing

Specific XA registers are reserved for use as 80C51 registers when translating code. The A register, the B register, and the data pointer all map to a pre-determined place in the XA register file (see figure 9.1). The accumulator (A) is the only one of these that required special hardware support in the XA, because the accumulator can be read or tested directly by certain instructions and in order to generate the parity flag.

The 4 banks of 8 byte registers that are found in the 80C51 are duplicated in the XA. The only difference is that in the XA, these registers do not normally overlap the lower 32 bytes of data memory space as they do in the 80C51. To allow code translation, a special 80C51 compatibility mode causes the XA register file to copy the 80C51 mapping to data memory. This mode is activated by the CM bit in the System Configuration Register (SCR).

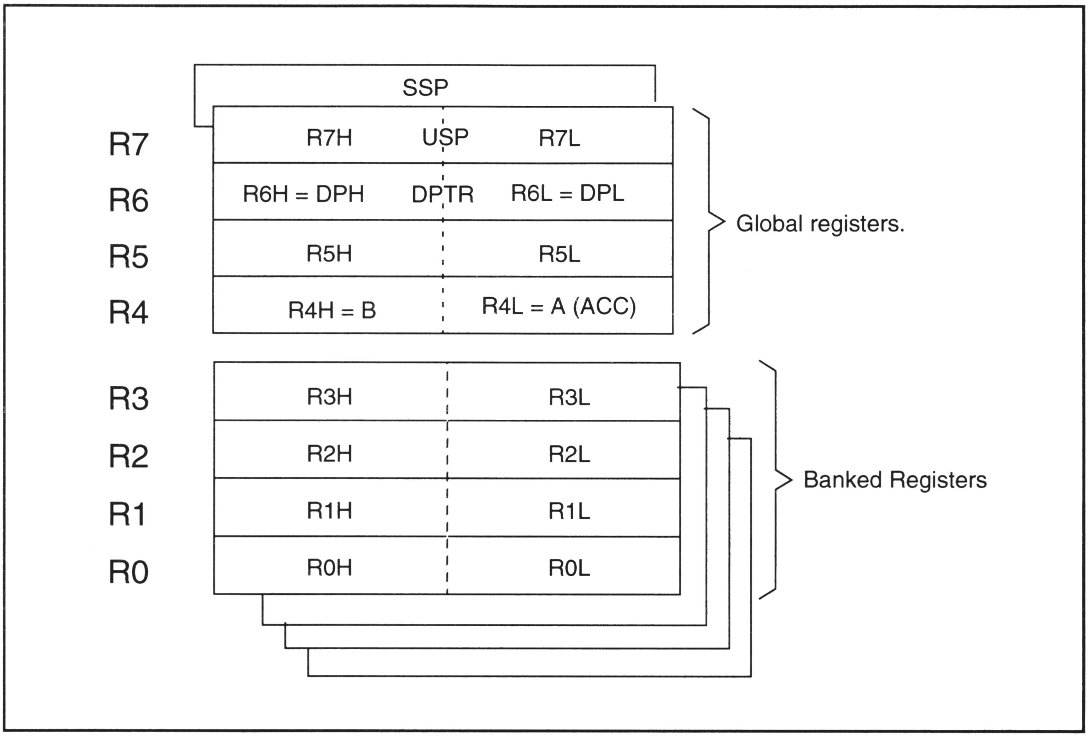

SSP
R7
R7H
USP
R7L
R6
R6H = DPH
DPTR
R6L = DPL
R5
R5H
R5L
R4
R4H = B
R4L = A (ACC)
Global registers.
R3
R3H
R3L
R2
R2H
R2L
R1
R1H
R1L
R0
R0H
R0L
Banked Registers

Figure 9.1. XA Register File

Other important registers of the 80C51 are provided in other ways. The program status word (PSW) of the XA is slightly different than the 80C51 PSW, so a special SFR address is reserved to provide an 80C51 compatible "view" of the PSW for use by translated code. This alternate PSW, called PSW51, is shown in the figure 9.2. The F0 flag and the F1 flag are simply readable and writable bits. The P flag provides an even parity bit for the 80C51 A register and always reflects the current contents of that register. Note that the P flag, the F0 flag, and the F1 flag only appear in the PSW51 register.

PSW51	C	AC	F0	RS1	RS0	OV	F1	P

Figure 9.2. PSW CPU status flags

The 80C51 indirect data memory access mode, using R0 or R1 as pointers, requires special support on the XA, where pointers are normally 16 bits in length. The 80C51 compatibility mode also causes the XA to mimic the 80C51 indirect scheme, using the first two bytes of the register file as indirect pointers, each zero extended to make a 16-bit address. Due to this and the previously mentioned register overlap to memory feature, the compatibility mode must be turned on in order to execute most translated 80C51 code on the XA.

The 80C51 mapped the special function registers (SFRs) into the direct address space, from address 80 hex to FF hex. SFRs were only accessed by instruction that contain the entire SFR address, so translation to the XA is fairly simple. Since references to SFRs are normally done by their name in 80C51 source code, the translation just copies the name into the XA code output. If an SFR happened to be referred to by its address, its name must be found so that it can be inserted into the XA code. This would require that an SFR table be available for the 80C51 derivative for which the code was originally written.

The XA has another mode which may be useful for translated 80C51 code. In order to save stack space as well as speed up execution, a Page Zero (PZ) mode causes return addresses on the stack to be saved as 16 bits only, instead of the usual 24 bits (which occupy 32 bits due to word alignment on the XA stack). All other program and data addresses are also forced to be 16-bits. If an entire 80C51 application program is translated to the XA, it will very likely fit within this 64K limit, allowing the use of this mode.

Other aspects of the processor stack have been altered on the XA. For one, the standard direction of stack growth for 16 bit processors has been adopted. So, the XA stack grows downward, from higher to lower addresses in data memory. The stack can now be nearly 64K in size if necessary, and begin anywhere in its data segment so may be easily moved to a new location for translated 80C51 applications. This stack direction change is important to match the stack contents to normal data memory accesses on the XA.

80C51 code translated to run on the XA will also tend to use more stack space for two reasons. First, the PSW is automatically saved during interrupt and exception processing on the XA. The original 80C51 code should have also saved the PSW explicitly, but the XA PSW is 16 bits in length. Secondly, the initial implementation of the XA allows only word writes to the stack. Both byte and word operations may be performed, but both types of operations use 16 bits of stack space.

The tendency for stack size increase, in addition to the stack growth direction will require some changes to be made if a complete 80C51 application program is translated to run on the XA.

9.1.2 Interrupt and Exception Processing

Interrupt handling on the XA is inherently much more powerful than it was on the 80C51. Along with this added power and flexibility comes some difference that must be taken into account for 80C51 code conversion.

Previously noted was the fact that the XA automatically saves the PSW during interrupt processing. If an 80C51 program relied on this not being the case somehow, it would not work without alteration. This type of reliance is not found in code using common programming practices and should be very rare.

The XA allows up to 15 interrupt priority levels, compared to only 2 in the standard 80C51, although up to 4 levels are available in a few of the newer 80C51 variations. These priorities are stored as 4-bit values, with the priority for 2 interrupts found in the same SFR byte. This is different (and much more powerful) than any 80C51 derivative, and will require minor changes to code that is translated.

The method of entering an interrupt routine in the XA uses a vector table stored in low addresses of the code memory. Each interrupt or exception source has a vector which consists of the address of the handler routine for that event and a new PSW value that is loaded when the vector is taken. This differs from the 80C51 approach of fixed addresses for the interrupt service routines, and again is a much more flexible and powerful method. So, if a complete 80C51 application program is converted for the XA, the interrupt service routines must be re-located above the XA vector table and the new address stored in the table, a very simple process.

9.1.3 On-Chip Peripherals

Compatibility with standard on-chip peripherals found in the 80C51 has been kept in the XA whenever possible and reasonable, but not to the extent that some enhancements are not made. The set of standard peripheral devices includes the UART, Timers 0 and 1, and Timer 2 from the 80C52.

The XA UART has been enhanced in a way that does not affect translated 80C51 code. Some additional features are added through the use of a new SFR, such as framing error detection, overrun detection, and break detection.

The timers remain the same except for one difference in the function, and a difference in timing. The functional change was to remove the 8048 timer mode (mode 0) and replace it with something much more useful: a 16-bit auto-reload mode. The relationship of timer count rates to the microcontroller oscillator has also been changed. This adds flexibility since this is now a programmable feature, allowing oscillator divided by 4, 16, or 64 to be used as the base count rate for all of the timers. Since XA performance is much higher (on a clock-by clock basis), an application converted to the XA from the 80C51 would likely not use the same oscillator frequency anyway.

9.1.4 Bus Interface

The customary 80C51 bus control signals are all found on the standard external XA bus. To provide the best performance, the details of some of these signals have changed somewhat, and a few new ones have been added. In addition to the well known ALE, $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{EA}}$, there are now also WAIT and $\overline{\text{WRH}}$. The WAIT signal causes wait states to be inserted into any XA bus cycle as long as it is asserted. The $\overline{\text{WRH}}$ signal is used to distinguish writes to the high order byte when the XA bus is configured to be 16 bits wide.

The multiplexed address/data bus has undergone some renovations on the XA as well. To get the most performance in a system executing code from the external bus, the XA separates the 4 least significant address lines on to their own pins. Since these lines normally change the most often, an ALE cycle would be required on every external code fetch if these lines were multiplexed as they are on the 80C51. The 80C51 had time to do this since its performance was not that high. The XA, however, uses only as many clocks as are needed to execute each instruction, so an ALE for every fetch would slow things down considerably. With this change, up to 16 bytes (or 8 words) of code may be accessed without the need to insert an ALE cycle on the XA bus.

The number of XA clocks used for each type of bus cycle (code read, data read, or data write) can also be programmed, so that slower peripheral devices can work with the XA without the need for an external WAIT state generator.

Due to the various changes to the bus just mentioned, an XA device cannot be completely pin compatible with an 80C51 derivative if the external bus is used. The changes to application hardware needed are relatively small and easy to make.

9.1.5 Instruction Set

The simplest goal of the XA for instruction set compatibility was to have every 80C51 instruction translate to one XA instruction. That has been achieved but for a single exception. The 80C51 instruction, XCHD or exchange digits, cannot be translated in that manner. XCHD is an instruction that is rarely used on the 80C51 and could not be implemented on the XA, due to its internal architecture, without adding a great deal of extra circuitry. So, if this instruction is encountered when 80C51 source code is being translated, a sequence of XA instructions is used to duplicate the function:

```
PUSH    R4H         ; Save temporary register.
MOV     R4H,(Ri)    ; Get second operand.
RR      R4H,#4      ; Swap one byte.
RR      R4L,#4      ; Swap second byte (the "A" register).
RL      R4,#4       ; Swap word.
                    ; Result is swapped nibbles in A and R4H.
MOV     (Ri),R4H    ; Store result.
POP     R4H         ; Restore temporary register.
```

If the application requires this sequence to not be interruptible, some additional instruction must be added in order to disable and re-enable interrupts. The table at the end of this section shows all of the other XA code replacements for 80C51 instructions.

The XA instruction set is much more powerful than the 80C51 instruction set, and as a direct consequence, the average number of bytes in an instruction is higher on the XA. In code written for the XA, the capability of a single instruction is high, so the size of an entire XA program will normally be smaller than the same program written for an 80C51. Of course, this depends on how much the application can take advantage of XA features. When code is translated from 80C51 source, however, the size change can be an issue.

In the case of a jump table, where the JMP @A+DPTR instruction is used to jump into a table of other jumps composed of the 80C51 AJMP instruction, the XA cannot always duplicate the function of the jumps in the table with instructions that are 2 bytes in length, as in the case of the AJMP instruction. An adjustment to the calculation of the table index will be required to make the translated code work properly. For a data table, accessed using MOVC @A+PC, the distance to the table may change, requiring a similar index adjustment.

Since the XA optimizes the timing of each instruction, there will be very little correspondence to the original 80C51 timing for the same code prior to translation to the XA. If the exact timing of a sequence of instructions is important to the application, the translated code must be altered, perhaps by adding NOPs or delay loops, to provide the necessary timing.

To show how a simple 80C51 to XA source code translator might work, a subroutine was extracted from a working 80C51 program and translated using the table at the end of this document and the other rules presented here. The original 80C51 source code was:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal: MOV     Temp2,A          ; Save step target for later use.
         MOV     B,#Steplow       ; Get low byte of step increment.
         MUL     AB               ; Multiply this by the step target.
         MOV     StepResult,B     ; Save high byte as partial result.
         MOV     Temp1,A          ; Save low byte to use for rounding.

         MOV     A,Temp2          ; Get back the step target.
         MOV     B,#StepHigh      ; Get high byte of step increment,
         MUL     AB               ; and multiply the two.

         ADD     A,StepResult     ; Add the two partial results.
         JNB     Temp1.7,Exit     ; Least significant byte > 80h?
         INC     A                ; If so, round up the final result.
Exit:    ADD     A,#MotorBot      ; Add in the 0 step displacement.
         MOV     StepResult,A     ; Save final step target.
         RET
```

The same code as translated for the XA is as follows:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal: MOV     Temp2,R4L        ; Save step target for later use.
         MOV     R4H,#Steplow     ; Get low byte of step increment.
         MULU.b  R4,R4H           ; Multiply this by the step target.
         MOV     StepResult,R4H ; Save high byte as partial result.
         MOV     Temp1,R4L        ; Save low byte to use for rounding.

         MOV     R4L,Temp2        ; Get back the step target.
         MOV     R4H,#StepHigh  ; Get high byte of step increment,
         MULU.b  R4,R4H           ; and multiply the two.

         ADD     R4L,StepResult ; Add the two partial results.
         JNB     Temp1.7,Exit     ; Least significant byte > 80h?
         ADDS    R4L,#1           ; If so, round up the final result.
Exit:    ADD     R4L,#MotorBot  ; Add in the 0 step displacement.
         MOV     StepResult,R4    ; Save final step target.
         RET
```

In this case, the translated code actually changed very little. Primarily, the 80C51 register names have been replaced by the new ones reserved for them in the XA. The increment (INC) instruction became a short add (ADDS), and the mnemonic for multiply (MUL) changed to MULU8.

Some basic statistical information about these code samples may be found in table 9.1. These statistics show a large performance increase for the XA code. This is significant because the code is only simple translated 80C51 code and therefore does not take any advantage of the XA's unique features.

Table 9.1: 80C51 to XA Code Translation Statistics

Statistic	**80C51 code**	**XA translation**	**Comments**
Code bytes	28	40	- one NOP added for branch alignment on XA
Clocks to execute	300	78	- includes XA pre-fetch queue analysis, raw execution is 66 clocks
Time to execute @ 20MHz	15 µsec	3.9 µsec	- a nearly 4x improvement without any optimization

9.2 Code Translation

Table 9.2 shows every 80C51 instruction type and the XA instruction that replaces it. An actual 80C51 to XA source code translator can make use of this table, but must also flag the compatibility exceptions noted in this section, so that any necessary adjustments may be made to the resulting XA source code.

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Arithmetic operations	
ADD A, Rn	ADD.b R, R
ADD A, #data8	ADD.b R, #data8
ADD A,dir8	ADD.b R, direct
ADD A, @Ri	ADD.b R, [R]
ADDC A, Rn	ADDC.bR, R
ADDC A, #data8	ADDC.bR, #data8
ADDC A,dir8	ADDC.bR, direct
ADDC A, @Ri	ADDC.bR, [R]
SUBB A, Rn	SUBB.bR, R
SUBB A, #data8	SUBB.bR, #data8
SUBB A, dir8	SUBB.bR, direct
SUBB A, @Ri	SUBB.bR, [R]
INC Rn	ADDS.bR, #1
INC dir8	ADDS.bdirect, #1
INC @Ri	ADDS.b[R], #1
INC A	ADDS.bR, #1
INC DPTR	ADDS.wR, #1
DEC Rn	ADDS.bR, #-1
DEC dir8	ADDS.bdirect, #-1
DEC @Ri	ADDS.b[R], #-1
DEC A	ADDS.bR, #-1
MUL AB	MULU.bR, R
DIV AB	DIVU.b R, R
DA A	DA R

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Logical operations	
ANL A, Rn	AND.b R, R
ANL A, #data8	AND.b R, #data8
ANL A, dir8	AND.b R, direct
ANL A, @Ri	AND.b R, [R]
ANL dir8, A	AND.b direct, R
ANL dir8, #data8	AND.b direct, #data8
ORL A, Rn	OR.b R, R
ORL A, #data8	OR.b R, #data8
ORL A, dir8	OR.b R, direct
ORL A, @Ri	OR.b R, [R]
ORL dir8, A	OR.b direct, R
ORL dir8, #data8	OR.b direct, #data8
XRL A, Rn	XOR.b R, R
XRL A, #data8	XOR.b R, #data8
XRL A, dir8	XOR.b R, direct
XRL A, @Ri	XOR.b R, [R]
XRL dir8, A	XOR.b direct, R
XRL dir8, #data8	XOR.b direct, #data8
CLR A	MOVS R, #0
CPL A	CPL.b R
SWAP A	RL.b R, #4
RL A	RL.b R, #1
RLC A	RLC.b R, #1
RR A	RR.b R, #1
RRC A	RRC.b R, #1
CLR C	CLR bit
CLR bit	CLR bit
SETB C	SETB bit
SETB bit	SETB bit
CPL C	XOR.b PSWL, #data8
CPL bit	XOR.b direct, #data8
ANL C, bit	AND C, bit
ANL C, /bit	AND C, /bit
ORL C, bit	OR C, bit
ORL C, /bit	OR C, /bit
MOV C, bit	MOV C, bit
MOV bit, C	MOV bit, C

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Data transfer	
MOV A, Rn	MOV.b R, R
MOV A, #data8	MOV.b R, #data8
MOV A, dir8	MOV.b R, direct
MOV A, @Ri	MOV.b R, [R]
MOV Rn, A	MOV.b R, R
MOV Rn, #data8	MOV.b R, #data8
MOV Rn, dir8	MOV.b R, direct
MOV dir8, A	MOV.b direct, R
MOV dir8, #data8	MOV.b direct, #data8
MOV dir8, Rn	MOV.b direct, R
MOV dir8, dir8	MOV.b direct, direct
MOV dir8, @Ri	MOV.b direct, [R]
MOV @Ri, A	MOV.b [R], R
MOV @Ri, dir8	MOV.b [R], direct
MOV @Ri, #data8	MOV.b [R], #data8
MOV DPTR, #data16	MOV.w R, #data16
XCH A, Rn	XCH.b R, R
XCH A, dir8	XCH.b R, direct
XCH A, @Ri	XCH.b R, R
XCHD A, @Ri	a sequence (see text)
PUSH dir8	PUSH.bdirect
POP dir8	POP.b direct
MOVX A, @Ri	MOVX.bR, [R]
MOVX A, @DPTR	MOVX.bR, [R]
MOVX @Ri, A	MOVX.b[R], R
MOVX @DPTR, A	MOVX.b[R], R
MOVC A, @A+DPTR	MOVC.bA, [A+DPTR]
MOVC A, @A+PC	MOVC.bA, [A+PC]

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Relative branches	
SJMP rel8	BR rel8
CJNE A, dir8, rel CJNE A, #data8, rel CJNE Rn, #data8, rel CJNE @Ri, #data8, rel	CJNE.b R, direct, rel CJNE.b R, #data8, rel CJNE.b R, #data8, rel CJNE.b [R], #data8, rel
DJNZ Rn, rel DJNZ dir8, rel	DJNZ.b R, rel DJNZ.b direct, rel
JZ rel JNZ rel JC rel JNC rel	JZ rel JNZ rel BCS rel BCC rel
Jumps, Calls, Returns, and Misc.	
NOP	NOP
AJMP addr11 LJMP addr16 JMP @A+DPTR	JMP rel16 JMP rel16 JUMP [A+DPTR]
ACALL addr11 LCALL addr16	CALL rel16 CALL rel16
RET RETI	RET RETI

9.3 New Instructions on the XA

While the XA instructions that are similar to 80C51 instructions have a larger addressing range, more status flags, etc., the XA also has many entirely new instructions and addressing modes that make writing new code for the XA much easier and more efficient. The new addressing modes also make the XA work very well with high level language compilers. A complete list of the new XA instructions and addressing modes in shown in table 9.3.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
alu.w ..., ...	All of the 80C51 arithmetic and logic instructions with a 16-bit data size.
SUBB R,...	Subtract (without borrow), all addressing modes.
alu [R], R	Arithmetic and logic operations (ADD, ADDC, SUB, SUBB, CMPAND, OR, XOR, and MOV) from a register to an indirect address.
alu R, [R+]	Arithmetic and logic operations from an indirect address to a register, with the indirect pointer automatically incremented.
alu R,[R+offset8/16]	Arith/Logic operations from an indirect offset address (with 8 or 16-bit offset) to a register.
alu direct, R	The 80C51 has only MOV direct, R.
alu [R], R	The 80C51 has only MOV [R], R.
alu [R+], R	Arith/Logic operations from a register to an indirect address, with the indirect pointer automatically incremented.
alu [R+offset8/16], R	Arith/Logic operations from a register to an indirect offset address (with 8 or 16-bit offset).
alu direct, #data8/16	Arith/Logic operations to a direct address with 8 or 16-bit immediate data.
alu [R], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data.
alu [R+], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data with the indirect pointer automatically incremented.
alu [R+offset8/16], #data8/16	Arith/Logic operations to an indirect offset address (with 8 or 16-bit offset), with 8 or 16-bit immediate data.
MOV direct, [R]	Move data from an indirect to a direct address.
ADDS R, #data4	The 80C51 can only increment or decrement a register by 1. ADDS has a range of +7 to -8.
ADDS [R], #data4	Add a short value to an indirect address.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
ADDS [R+], #data4	Add a short value to an indirect offset address, with the indirect pointer automatically incremented.
ADDS [R+offset8/16], #data4	Add a short value to an indirect offset address (with 8 or 16-bit offset).
ADDS direct, #data4	Add a short value to a direct address.
MOVS ..., #data4	Move short data to destination using any of the same addressing modes as ADDS.
ASL R, R	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from register.
ASR R, R	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from register.
LSR R, R	Logical shift right a byte, word, or double word, up to 31 places, shift count read from register.
ASL R, #DATA4/5	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from instruction.
ASR R, #DATA4/5	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
LSR R, #DATA4/5	Logical shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
DIV R, R	Signed divide of 32 bits register by 16 bit register, or 16 bit register by 8 bit register.
DIVU R, R	Unsigned divide of 32 bit register by 16 bit register, or 16 bit register by 8 bit register.
MUL R, R	Signed multiply of 16 bit register by 16 bit register, or 8 bit register by 8 bit register.
MULU R, R	Unsigned multiply of 16 bit register by 16 bit register.
DIV R, #data8/16	Signed divide of 32 bits register by 16 bit immediate, or 16 bit register by 8 bit immediate.
DIVU R, #data8/16	Unsigned divide of 32 bit register by 16 bit immediate, or 16 bit register by 8 bit immediate.
MUL R, #data8/16	Signed multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
MULU R, #data8/16	Unsigned multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.
LEA R, R+offset8/16	Load effective address, duplicates the offset8 or 16-bit addressing mode calculation but saves the address in a register.
NEG R	Negate, performs a twos complement operation on a register.
SEXT R	Sign extend, copies the sign flag from the last operation into an 8 or 16-bit register.
NORM R, R	Normalize. Shifts a byte, word, or double word register left until the MSB becomes a 1. The number of shifts used is stored in a register.
RL, RR, RLC, RRC R,#data4	All of the 80C51 rotate modes with 16-bit data size and a variable number of bit positions (up to 15 places).
MOV [R+], [R+]	Block move. Move data from an indirect address to another indirect address, incrementing both pointers.
MOV R, USP and USP, R	Allows system code to move a value to or from the user stack pointer. Handy in multi-tasking applications.
MOVC R, [R+]	Move data from an indirect address in the code space to a register, with the indirect pointer automatically incremented.
PUSH and POP Rlist	PUSH and POP up to 8 word registers in one instruction.
PUSHU and POPU Rlist or direct	Allows system code to write to or read the user stack. Handy in multi-tasking applications.
conditional branches	A complete set of conditional branches, including BEQ, BNE, BG, BGE, BGT, BL, BLE, BMI, BPL, BNV, and BOV.
CALL [R]	Call indirect, to an address contained in a register.
CALL rel16	Call anywhere in a +/- 64K range.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
FCALL addr24	Far call, anywhere within the XA 16Mbyte code address space.
JMP [R]	Jump indirect, to an address contained in a register.
JMP rel16	Jump anywhere in a +/- 64K range.
FJMP addr24	Far jump, anywhere within the XA 16Mbyte code address space.
JMP [[R+]]	Jump double indirect with auto-increment. Used to branch to a sequence of addresses contained in a table.
BKPT	Breakpoint, a debugging feature.
RESET	Allows software to completely reset the XA in one instruction.
TRAP #data4	Call one of up to 16 system services. Acts like an immediate interrupt.

Section 3

XA Family Derivatives

CONTENTS

XA-G1 CMOS single-chip 16-bit microcontroller 317
XA-G2 CMOS single-chip 16-bit microcontroller 346
XA-G3 CMOS single-chip 16-bit microcontroller 375

CMOS single-chip 16-bit microcontroller

XA-G1

FAMILY DESCRIPTION

The Philips Semiconductors XA (eXtended Architecture) family of 16-bit single-chip microcontrollers is powerful enough to easily handle the requirements of high performance embedded applications, yet inexpensive enough to compete in the market for high-volume, low-cost applications.

The XA family provides an upward compatibility path for 80C51 users who need higher performance and 64k or more of program memory. Existing 80C51 code can also be easily be translated to run on XA microcontrollers.

The performance of the XA architecture supports the comprehensive bit-oriented operations of the 80C51 while incorporating support for multi-tasking operating systems and high-level languages such as C. The speed of the XA architecture, at 10 to 100 times that of the 80C51, gives designers and easy path to truly high performance embedded control.

The XA architecture supports:

- Upward compatibility with the 80C51 architecture
- 16-bit fully static CPU with a 24-bit program and data address range
- Eight 16-bit CPU registers each capable of performing all arithmetic and logic operations as well as acting as memory pointers. Operations may also be performed directly to memory.
- Both 8-bit and 16-bit CPU registers, each capable of performing all arithmetic and logic operations.
- An enhanced instruction set that includes bit intensive logic operations and fast signed or unsigned 16 × 16 multiply and 32 / 16 divide
- Instruction set tailored for high level language support
- Multi-tasking and real-time executives that include up to 32 vectored interrupts, 16 software traps, segmented data memory, and banked registers to support context switching
- Low power operation, which is intrinsic to the XA architecture, includes power-down and idle modes.

More detailed information on the core is available in the XA User Guide.

SPECIFIC FEATURES OF THE XA-G1

- 20-bit address range, 1 megabyte each program and data space. (Note that the XA architecture supports up to 24 bit addresses.)
- 2.7V to 5.5V operation
- 8K bytes on-chip EPROM/ROM program memory
- 512 bytes of on-chip data RAM
- Three counter/timers with enhanced features (equivalent to 80C51 T0, T1, and T2)
- Watchdog timer
- Two enhanced UARTs
- Four 8-bit I/O ports with 4 programmable output configurations
- 44-pin PLCC and 44-pin LQFP packages

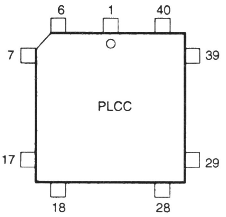
ORDERING INFORMATION

ROM	EPROM[1]		TEMPERATURE RANGE °C AND PACKAGE	FREQ (MHz)	DRAWING NUMBER
P51XAG13KB BD	P51XAG17KB BD	OTP	0 to +70, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG13KB A	P51XAG17KB A	OTP	0 to +70, Plastic Leaded Chip Carrier	30	SOT187–2
	P51XAG17KB KA	UV	0 to +70, Ceramic Leaded Chip Carrier	30	1472A
P51XAG13KF BD	P51XAG17KF BD	OTP	–40 to +85, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG13KF A	P51XAG17KF A	OTP	–40 to +85, Plastic Leaded Chip Carrier	30	SOT187–2
	P51XAG17KF KA	UV	–40 to +85, Ceramic Leaded Chip Carrier	30	1472A

NOTE:
1. OTP = One Time Programmable EPROM. UV = Erasable EPROM.

PIN CONFIGURATIONS

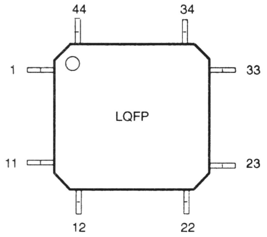
44-Pin PLCC Package



Pin	Function	Pin	Function
1	V_{SS}	23	V_{DD}
2	P1.0/A0/$\overline{\text{WRH}}$	24	P2.0/A12D8
3	P1.1/A1	25	P2.1/A13D9
4	P1.2/A2	26	P2.2/A14D10
5	P1.3/A3	27	P2.3/A15D11
6	P1.4/RxD1	28	P2.4/A16D12
7	P1.5/TxD1	29	P2.5/A17D13
8	P1.6/T2	30	P2.6/A18D14
9	P1.7/T2EX	31	P2.7/A19D15
10	$\overline{\text{RST}}$	32	$\overline{\text{PSEN}}$
11	P3.0/RxD0	33	ALE/$\overline{\text{PROG}}$
12	NC	34	NC
13	P3.1/TxD0	35	$\overline{\text{EA}}$/V_{PP}/WAIT
14	P3.2/$\overline{\text{INT0}}$	36	P0.7/A11D7
15	P3.3/$\overline{\text{INT1}}$	37	P0.6/A10D6
16	P3.4/T0	38	P0.5/A9D5
17	P3.5/T1/BUSW	39	P0.4/A8D4
18	P3.6/$\overline{\text{WRL}}$	40	P0.3/A7D3
19	P3.7/$\overline{\text{RD}}$	41	P0.2/A6D2
20	XTAL2	42	P0.1/A5D1
21	XTAL1	43	P0.0/A4D0
22	V_{SS}	44	V_{DD}

SU00525

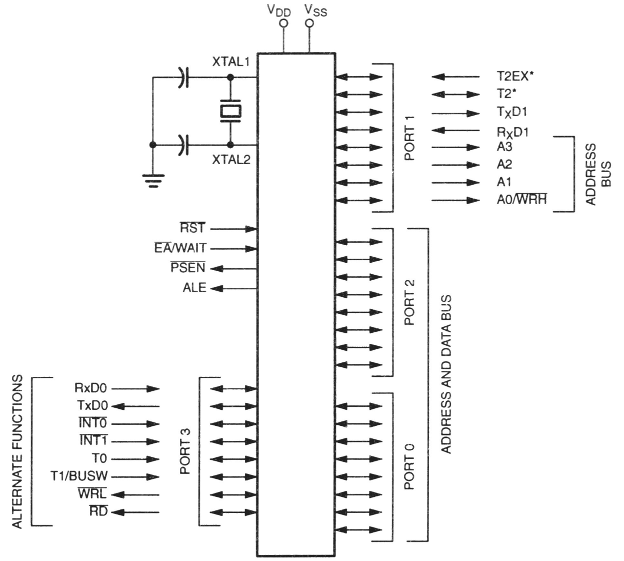
44-Pin LQFP Package



Pin	Function	Pin	Function
1	P1.5/TxD1	23	P2.5/A17D13
2	P1.6/T2	24	P2.6/A18D14
3	P1.7/T2EX	25	P2.7/A19D15
4	$\overline{\text{RST}}$	26	$\overline{\text{PSEN}}$
5	P3.0/RxD0	27	ALE/$\overline{\text{PROG}}$
6	NC	28	NC
7	P3.1/TxD0	29	$\overline{\text{EA}}$/V_{PP}/WAIT
8	P3.2/$\overline{\text{INT0}}$	30	P0.7/A11D7
9	P3.3/$\overline{\text{INT1}}$	31	P0.6/A10D6
10	P3.4/T0	32	P0.5/A9D5
11	P3.5/T1/BUSW	33	P0.4/A8D4
12	P3.6/$\overline{\text{WRL}}$	34	P0.3/A7D3
13	P3.7/$\overline{\text{RD}}$	35	P0.2/A6D2
14	XTAL2	36	P0.1/A5D1
15	XTAL1	37	P0.0/A4D0
16	V_{SS}	38	V_{DD}
17	V_{DD}	39	V_{SS}
18	P2.0/A12D8	40	P1.0/A0/$\overline{\text{WRH}}$
19	P2.1/A13D9	41	P1.1/A1
20	P2.2/A14D10	42	P1.2/A2
21	P2.3/A15D11	43	P1.3/A3
22	P2.4/A16/D12	44	P1.4/RxD1

SU00580

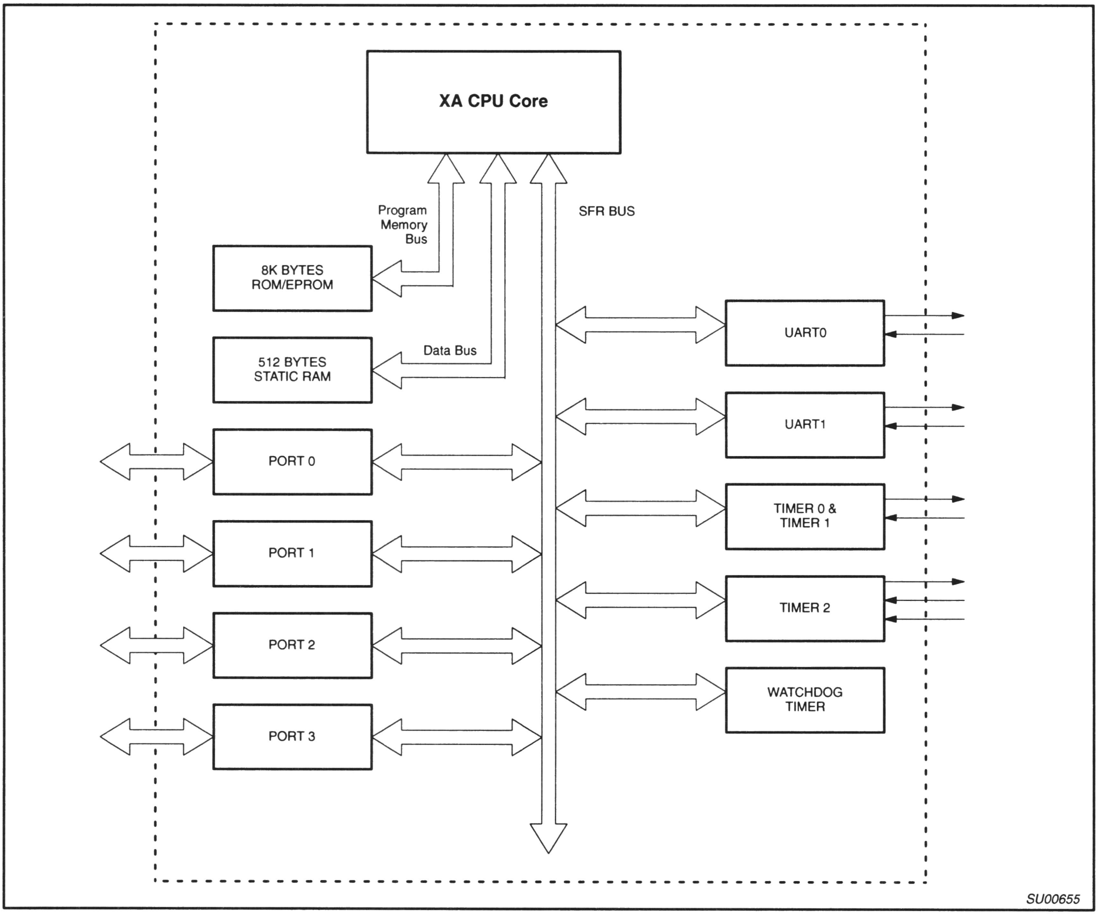
LOGIC SYMBOL


V_{DD}
V_{SS}
XTAL1
XTAL2
RST
EA/WAIT
PSEN
ALE
PORT 1
T2EX*
T2*
T_XD1
R_XD1
A3
A2
A1
A0/WRH
ADDRESS BUS
PORT 2
ADDRESS AND DATA BUS
PORT 0
PORT 3
ALTERNATE FUNCTIONS
RxD0
TxD0
INT0
INT1
T0
T1/BUSW
WRL
RD

* NOT AVAILABLE ON 40-PIN DIP PACKAGE

SU00526

BLOCK DIAGRAM


XA CPU Core
Program Memory Bus
SFR BUS
8K BYTES ROM/EPROM
512 BYTES STATIC RAM
Data Bus
PORT 0
PORT 1
PORT 2
PORT 3
UART0
UART1
TIMER 0 & TIMER 1
TIMER 2
WATCHDOG TIMER
SU00655

PIN DESCRIPTIONS

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
V_{SS}	1, 22	16	I	**Ground:** 0V reference.
V_{DD}	23, 44	17	I	**Power Supply:** This is the power supply voltage for normal, idle, and power down operation.
P0.0 – P0.7	43–36	37–30	I/O	**Port 0:** Port 0 is an 8-bit I/O port with a user-configurable output type. Port 0 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 0 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used, Port 0 becomes the multiplexed low data/instruction byte and address lines 4 through 11. Port 0 also outputs the code bytes during program verification and receives code bytes during EPROM programming.
P1.0 – P1.7	2–9	40–44, 1–3	I/O	**Port 1:** Port 1 is an 8-bit I/O port with a user-configurable output type. Port 1 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 1 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 1 also provides special functions as described below.
	2	40	O	**A0/$\overline{\mathrm{WRH}}$:** Address bit 0 of the external address bus when the external data bus is configured for an 8 bit width. When the external data bus is configured for a 16 bit width, this pin becomes the high byte write strobe.
	3	41	O	**A1:** Address bit 1 of the external address bus.
	4	42	O	**A2:** Address bit 2 of the external address bus.
	5	43	O	**A3:** Address bit 3 of the external address bus.
				Port 1 also provides various special functions as described below.
	6	44	I	**RxD1 (P1.4):** Receiver input for serial port 1.
	7	1	O	**TxD1 (P1.5):** Transmitter output for serial port 1.
	8	2	I	**T2 (P1.6):** Timer/counter 2 external count input/clockout.
	9	3	I	**T2EX (P1.7):** Timer/counter 2 reload/capture/direction control
P2.0 – P2.7	24–31	18–25	I/O	**Port 2:** Port 2 is an 8-bit I/O port with a user-configurable output type. Port 2 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 2 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used in 16-bit mode, Port 2 becomes the multiplexed high data/instruction byte and address lines 12 through 19. When the external program/data bus is used in 8-bit mode, the number of address lines that appear on port 2 is user programmable. Port 2 also receives the low-order address byte during program memory verification.
P3.0 – P3.7	11, 13–19	5, 7–13	I/O	**Port 3:** Port 3 is an 8-bit I/O port with a user configurable output type. Port 3 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. the operation of port 3 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 3 pins receive the high order address bits during EPROM programming and verification. Port 3 also provides various special functions as described below.
	11	5	I	**RxD0 (P3.0):** Receiver input for serial port 0.
	13	7	O	**TxD0 (P3.1):** Transmitter output for serial port 0.
	14	8	I	**$\overline{\mathrm{INT0}}$ (P3.2):** External interrupt 0 input.
	15	9	I	**$\overline{\mathrm{INT1}}$ (P3.3):** External interrupt 1 input.
	16	10	I/O	**T0 (P3.4):** Timer 0 external input, or timer 0 overflow output.
	17	11	I/O	**T1/BUSW (P3.5):** Timer 1 external input, or timer 1 overflow output. The value on this pin is latched as the external reset input is released and defines the default external data bus width (BUSW).
	18	12	O	**$\overline{\mathrm{WRL}}$ (P3.6):** External data memory low byte write strobe.
	19	13	O	**$\overline{\mathrm{RD}}$ (P3.7):** External data memory read strobe.

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
$\overline{\text{RST}}$	10	4	I	**Reset:** A low on this pin resets the microcontroller, causing I/O ports and peripherals to take on their default states, and the processor to begin execution at the address contained in the reset vector. Refer to the section on Reset for details.
ALE/$\overline{\text{PROG}}$	33	27	I/O	**Address Latch Enable/Program Pulse:** A high output on the ALE pin signals external circuitry to latch the address portion of the multiplexed address/data bus. A pulse on ALE occurs only when it is needed in order to process a bus cycle. During EPROM programming, this pin is used as the program pulse input.
$\overline{\text{PSEN}}$	32	26	O	**Program Store Enable:** The read strobe for external program memory. When the microcontroller accesses external program memory, $\overline{\text{PSEN}}$ is driven low in order to enable memory devices. $\overline{\text{PSEN}}$ is only active when external code accesses are performed.
$\overline{\text{EA}}$/WAIT/ V_{PP}	35	29	I	**External Access/Wait/Programming Supply Voltage:** The $\overline{\text{EA}}$ input determines whether the internal program memory of the microcontroller is used for code execution. The value on the $\overline{\text{EA}}$ pin is latched as the external reset input is released and applies during later execution. When latched as a 0, external program memory is used exclusively, when latched as a 1, internal program memory will be used up to its limit, and external program memory used above that point. After reset is released, this pin takes on the function of bus Wait input. If Wait is asserted high during any external bus access, that cycle will be extended until Wait is released. During EPROM programming, this pin is also the programming supply voltage input.
XTAL1	21	15	I	**Crystal 1:** Input to the inverting amplifier used in the oscillator circuit and input to the internal clock generator circuits.
XTAL2	20	14	O	**Crystal 2:** Output from the oscillator amplifier.

SPECIAL FUNCTION REGISTERS

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
BCR	Bus configuration register	46A	—	—	—	WAITD	BUSD	BC2	BC1	BC0	Note 1
BTRH	Bus timing register low byte	469	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0	FF
BTRL	Bus timing register high byte	468	WM1	WM0	ALEW	—	CR1	CR0	CRA1	CRA0	EF
CS	Code segment	443									00
DS	Data segment	441									00
ES	extra segment	442									00
			33F	33E	33D	33C	33B	33A	339	338	
IEH*	Interrupt enable high byte	427	—	—	—	—	ETI1	ERI1	ETI0	ERI0	00
			337	336	335	334	333	332	331	330	
IEL*	Interrupt enable low byte	426	EA	—	—	ET2	ET1	EX1	ET0	EX0	00
IPA0	Interrupt priority 0	4A0	—	PT0			—	PX0			00
IPA1	Interrupt priority 1	4A1	—	PT1			—	PX1			00
IPA2	Interrupt priority 2	4A2	—	—			—	PT2			00
IPA4	Interrupt priority 4	4A4	—	PTI0			—	PRI0			00
IPA5	Interrupt priority 5	4A5	—	PTI1			—	PRI1			00
			387	386	385	384	383	382	381	380	
P0*	Port 0	430	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FF
			38F	38E	38D	38C	38B	38A	389	388	
P1*	Port 1	431	T2EX	T2	P1.5	P1.4	P1.3	P1.2	P1.1	WR1	FF
			397	396	395	394	393	392	391	390	
P2*	Port 2	432	P2.7	P2.6	P2.5	P2.4	TxD1	RxD1	P2.1	P2.0	FF

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
			39F	39E	39D	39C	39B	39A	399	398	
P3*	Port 3	433	RD	WR	T1	T0	INT1	INT0	TxD0	RxD0	FF
P0CFGA	Port 0 configuration A	470									Note 5
P1CFGA	Port 1 configuration A	471									Note 5
P2CFGA	Port 2 configuration A	472									Note 5
P3CFGA	Port 3 configuration A	473									Note 5
P0CFGB	Port 0 configuration B	4F0									Note 5
P1CFGB	Port 1 configuration B	4F1									Note 5
P2CFGB	Port 2 configuration B	4F2									Note 5
P3CFGB	Port 3 configuration B	4F3									Note 5
			227	226	225	224	223	222	221	220	
PCON*	Power control register	404	—	—	—	—	—	—	PD	IDL	00
			20F	20E	20D	20C	20B	20A	209	208	
PSWH*	Program status word (high byte)	401	SM	TM	RS1	RS0	IM3	IM2	IM1	IM0	Note 2
			207	206	205	204	203	202	201	200	
PSWL*	Program status word (low byte)	400	C	AC	—	—	—	V	N	Z	Note 2
			217	216	215	214	213	212	211	210	
PSW51*	80C51 compatible PSW	402	C	AC	F0	RS1	RS0	V	F1	P	Note 3
RTH0	Timer 0 extended reload, high byte	455									00
RTH1	Timer 1 extended reload, high byte	457									00
RTL0	Timer 0 extended reload, low byte	454									00
RTL1	Timer 1 extended reload, low byte	456									00
			307	306	305	304	303	302	301	300	
S0CON*	Serial port 0 control register	420	SM0_0	SM1_0	SM2_0	REN_0	TB8_0	RB8_0	TI_0	RI_0	00
			30F	30E	30D	30C	30B	30A	309	308	
S0STAT*	Serial port 0 extended status	421	—	—	—	—	FE0	BR0	OE0	STINT0	00
S0BUF	Serial port 0 buffer register	460									x
S0ADDR	Serial port 0 address register	461									00
S0ADEN	Serial port 0 address enable register	462									00
			327	326	325	324	323	322	321	320	
S1CON*	Serial port 1 control register	424	SM0_1	SM1_1	SM2_1	REN_1	TB8_1	RB8_1	TI_1	RI_1	00
			32F	32E	32D	32C	32B	32A	329	328	
S1STAT*	Serial port 1 extended status	425	—	—	—	—	FE1	BR1	OE1	STINT1	00
S1BUF	Serial port 1 buffer register	464									x
S1ADDR	Serial port 1 address register	465									00
S1ADEN	Serial port 1 address enable register	466									00
SCR	System configuration register	440	—	—	—	—	PT1	PT0	CM	PZ	00
			21F	21E	21D	21C	21B	21A	219	218	
SSEL*	Segment selection register	403	ESWEN	R6SEG	R5SEG	R4SEG	R3SEG	R2SEG	R1SEG	R0SEG	00
SWE	Software Interrupt Enable	47A	—	SWE7	SWE6	SWE5	SWE4	SWE3	SWE2	SWE1	00

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
			357	356	355	354	353	352	351	350	
SWR*	Software Interrupt Request	42A	—	SWR7	SWR6	SWR5	SWR4	SWR3	SWR2	SWR1	00
			2C7	2C6	2C5	2C4	2C3	2C2	2C1	2C0	
T2CON*	Timer 2 control register	418	TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2	00
			2CF	2CE	2CD	2CC	2CB	2CA	2C9	2C8	
T2MOD*	Timer 2 mode control	419	—	—	RCLK1	TCLK1	—	T2RD	T2OE	DCEN	00
TH2	Timer 2 high byte	459									00
TL2	Timer 2 low byte	458									00
T2CAPH	Timer 2 capture register, high byte	45B									00
T2CAPL	Timer 2 capture register, low byte	45A									00
			287	286	285	284	283	282	281	280	
TCON*	Timer 0 and 1 control register	410	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00
TH0	Timer 0 high byte	451									00
TH1	Timer 1 high byte	453									00
TL0	Timer 0 low byte	450									00
TL1	Timer 1 low byte	452									00
TMOD	Timer 0 and 1 mode register	45C	GATE	C/T	M1	M0	GATE	C/T	M1	M0	00
			28F	28E	28D	28C	28B	28A	289	288	
TSTAT*	Timer 0 and 1 extended status	411	—	—	—	—	T1RD	T1OE	T0RD	T0OE	00
			2FF	2FE	2FD	2FC	2FB	2FA	2F9	2F8	
WDCON*	Watchdog control register	41F	PRE2	PRE1	PRE0	—	—	WDRUN	WDTOF	—	Note 6
WDL	Watchdog timer reload	45F									00
WFEED1	Watchdog feed 1	45D									x
WFEED2	Watchdog feed 2	45E									x

NOTES:

* SFRs are bit addressable.
1. At reset, the BCR register is loaded with the binary value 0000 0a11, where "a" is the value on the BUSW pin.
2. SFR is loaded from the reset vector.
3. All bits except F1, F0, and P are loaded from the reset vector. Those bits are all 0.
4. Unimplemented bits in SFRs are X (unknown) at all times. Ones should not be written to these bits since they may be used for other purposes in future XA derivatives. The reset value shown for these bits is 0.
5. Port configurations default to quasi-bidirectional when the XA begins execution from internal code memory after reset, based on the condition found on the EA pin. Thus all PnCFGA registers will contain FF and PnCFGB registers will contain 00. When the XA begins execution using external code memory, the default configuration for pins that are associated with the external bus will be push-pull. The PnCFGA and PnCFGB register contents will reflect this difference.
6. The WDCON reset value is E6 for a Watchdog reset, E4 for all other reset causes.

XA-G1 TIMER/COUNTERS

The XA has two standard 16-bit enhanced Timer/Counters: Timer 0 and Timer 1. Additionally, it has a third 16-bit Up/Down timer/counter, T2. A central timing generator in the XA core provides the time-base for all XA Timers and Counters. The timer/event counters can perform the following functions:

- Measure time intervals and pulse duration
- Count external events
- Generate interrupt requests
- Generate PWM or timed output waveforms

All of the XA-G1 timer/counters (Timer 0, Timer 1 and Timer 2) can be independently programmed to operate either as timers or event counters via the C/T bit in the TnCON register. These timers may be dynamically read during program execution.

The base clock rate of all of the XA-G1 timers is user programmable. This applies to timers T0, T1, and T2 when running in timer mode (as opposed to counter mode), and the watchdog timer. The clock driving the timers is called TCLK and is determined by the setting of two bits (PT1, PT0) in the System Configuration Register (SCR). The frequency of TCLK may be selected to be the oscillator input divided by 4 (Osc/4), the oscillator input divided by 16 (Osc/16), or the oscillator input divided by 64 (Osc/64). This gives a range of possibilities for the XA timer functions, including baud rate generation, Timer 2 capture. Note that this single rate setting applies to all of the timers.

When timers T0, T1, or T2 are used in the counter mode, the register will increment whenever a falling edge (high to low transition) is detected on the external input pin corresponding to the timer clock. These inputs are sampled once every 2 oscillator cycles, so it can take as many as 4 oscillator cycles to detect a transition. Thus the maximum count rate that can be supported is Osc/4. The duty cycle of the timer clock inputs is not important, but any high or low state on the timer clock input pins must be present for 2 oscillator cycles before it is guaranteed to be “seen” by the timer logic.

Timer 0 and Timer 1

The “Timer” or “Counter” function is selected by control bits C/T in the special function register TMOD. These two Timer/Counters have four operating modes, which are selected by bit-pairs (M1, M0) in the TMOD register. Timer modes 1, 2, and 3 in XA are kept identical to the 80C51 timer modes for code compatibility. Only the mode 0 is replaced in the XA by a more powerful 16-bit auto-reload mode. This will give the XA timers a much larger range when used as time bases.

The recommended M1, M0 settings for the different modes are shown in Figure 2.

SCR Address:440
Not Bit Addressable
Reset Value: 00H

MSB LSB

—	—	—	—	PT1	PT0	CM	PZ

PT1	PT0	OPERATING
		Prescaler selection.
0	0	Osc/4
0	1	Osc/16
1	0	Osc/64
1	1	Reserved
CM		Compatibility Mode allows the XA to execute most translated 80C51 code on the XA. The XA register file must copy the 80C51 mapping to data memory and mimic the 80C51 indirect addressing scheme.
PZ		Page Zero mode forces all program and data addresses to 16-bits only. This saves stack space and speeds up execution but limits memory access to 64k.

SU00589

Figure 1. System Configuration Register (SCR)

TMOD Address:45C
Not Bit Addressable
Reset Value: 00H

MSB							LSB
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
TIMER 1				**TIMER 0**			

GATE Gating control when set. Timer/Counter "n" is enabled only while "$\overline{\text{INTn}}$" pin is high and "TRn" control bit is set. When cleared Timer "n" is enabled whenever "TRn" control bit is set.

C/$\overline{T}$ Timer or Counter Selector cleared for Timer operation (input from internal system clock.) Set for Counter operation (input from "Tn" input pin).

M1	**M0**	**OPERATING**
0	0	16-bit auto-reload timer/counter
0	1	16-bit non-auto-reload timer/counter
1	0	8-bit auto-reload timer/counter
1	1	Dual 8-bit timer mode (timer 0 only)

SU00605

Figure 2. Timer/Counter Mode Control (TMOD) Register

New Enhanced Mode 0

For timers T0 or T1 the 13-bit count mode on the 80C51 (current Mode 0) has been replaced in the XA with a 16-bit auto-reload mode. Four additional 8-bit data registers (two per timer: RTHn and RTLn) are created to hold the auto-reload values. In this mode, the TH overflow will set the TF flag in the T2CON register and cause both the TL and TH counters to be loaded from the RTL and RTH registers respectively.

These new SFRs will also be used to hold the TL reload data in the 8-bit auto-reload mode (Mode 2) instead of TH.

Mode 1

Mode 1 is the 16-bit non-auto reload mode.

Mode 2

Mode 2 configures the Timer register as an 8-bit Counter (TLn) with automatic reload. Overflow from TLn not only sets TFn, but also reloads TLn with the contents of RTLn, which is preset by software. The reload leaves THn unchanged.

Mode 2 operation is the same for Timer/Counter 0.

Mode 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting TR1 = 0.

Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. TL0 uses the Timer 0 control bits: C/T, GATE, TR0, INT0, and TF0. TH0 is locked into a timer function and takes over the use of TR1 and TF1 from Timer 1. Thus, TH0 now controls the "Timer 1" interrupt.

Mode 3 is provided for applications requiring an extra 8-bit timer. When Timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt.

TCON Address:441
Bit Addressable
Reset Value: 00H

MSB							LSB
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

BIT	**SYMBOL**	**FUNCTION**
TCON.7	TF1	Timer 1 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.6	TR1	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TCON.5	TF0	Timer 0 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.4	TR0	Timer 0 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TCON.3	IE1	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.2	IT1	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
TCON.1	IE0	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.0	IT0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

SU00604

Figure 3. Timer/Counter Control (TCON) Register

T2CON Address:418
Bit Addressable
Reset Value: 00H

MSB LSB

TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2

BIT	SYMBOL	FUNCTION
T2CON.7	TF2	Timer 2 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or clearing the bit in software. TF2 will not be set when RCLK0, RCLK1, TCLK0, TCLK1 or T2OE=1.
T2CON.6	EXF2	Timer 2 external flag is set when a capture or reload occurs due to a negative transition on T2EX (and EXEN is set). This flag will cause a Timer 2 interrupt when this interrupt is enabled. EXF2 is cleared by software.
T2CON.5	RCLK0	Receive Clock Flag.
T2CON.4	TCLK0	Transmit Clock Flag. RCLK0 and TCLK0 are used to select Timer 2 overflow rate as a clock source for UART0.
T2CON.3	EXEN2	Timer 2 external enable flag allows a capture or reload to occure due to a negative transition on T2EX.
T2CON.2	TR2	Start=1/Stop=0 control for Timer 2.
T2CON.1	C/T2	Timer or counter select. 0=Internal timer 1=External event counter (falling edge triggered)
T2CON.0	IT0	Capture/Reload flag. If CP/RL2 & EXEN2=1 captures will occur on negative transitions of T2EX. If CP/RL2=0, EXEN2=1 auto reloads occur with either Timer 2 overflows or negative transitions at T2EX. If RCLK or TCLK=1 the timer is set to auto reload on Timer 2 overflow, this bit has no effect.

SU00606

Figure 4. Timer/Counter 2 Control (T2CON) Register

New Timer-Overflow Toggle Output

In the XA, the timer module now has two outputs, which toggle on overflow from the individual timers. The same device pins that are used for the T0 and T1 count inputs are also used for the new overflow outputs. An SFR bit (TnOE in the TSTAT register) is associated with each counter and indicates whether Port-SFR data or the overflow signal is output to the pin. These outputs could be used in applications for generating variable duty cycle PWM outputs (changing the auto-reload register values). Also variable frequency (Osc/8 to Osc/8,388,608) outputs could be achieved by adjusting the prescaler along with the auto-reload register values. With a 30.0MHz oscillator, this range would be 3.58Hz to 3.75MHz.

Timer T2

This is a 16-bit up or down counter, which can be operated as either a timer or event counter. It can be operated in one of three different modes (autoreload, capture or as the baud rate generator for either or both UARTs).

In the autoreload mode the Timer can be set to count up or down by setting or clearing the bit DCEN in the T2MOD Special Function Register. The SFR's T2CAPH and T2CAPL are used to reload the Timer upon overflow or to capture a 1-to-0 transition on the T2EX input (P1.7).

In the Capture mode Timer 2 can either set TF2 and generate an interrupt or capture its value. To capture Timer 2 in response to a 1-to-0 transition on the T2EX input, the EXEN2 bit in the T2CON must be set. Timer 2 is then captured in SFR's T2CAP2H and T2CAP2L.

As the baud rate generator, Timer 2 is selected by setting one of the RCLK and/or TCLK bits in T2CON or T2MOD. As the baud rate generator Timer 2 is incremented by TCLK.

Programmable Clock-Out

A 50% duty cycle clock can be programmed to come out on P1.6. This pin, besides being a regular I/O pin, has two alternate functions. It can be programmed (1) to input the external clock for Timer/Counter 2 or (2) to output a 50% duty cycle clock ranging from 3.58Hz to 3.75MHz at a 30MHz operating frequency.

To configure the Timer/Counter 2 as a clock generator, bit C/T2 (in T2CON) must be cleared and bit T20E in T2MOD must be set. Bit TR2 (T2CON.2) also must be set to start the timer.

The Clock-Out frequency depends on the oscillator frequency and the reload value of Timer 2 capture registers (TCAP2H, TCAP2L) as shown in this equation:

$$\frac{\text{TCLK}}{2 \times (65536\text{–TCAP2H, TCAP2L})}$$

In the Clock-Out mode Timer 2 roll-overs will not generate an interrupt. This is similar to when it is used as a baud-rate generator. It is possible to use Timer 2 as a baud-rate generator and a clock generator simultaneously. Note, however, that the baud-rate and the Clock-Out frequency will be the same.

TSTAT Address:411
Bit Addressable
Reset Value: 00H

MSB LSB

—	—	—	—	T1RD	T1OE	T0RD	T0OE

BIT	SYMBOL	FUNCTION
TSTAT.3	T1RD	This bit is set, for Timer 1, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
TSTAT.2	T1OE	When 0, this bit allows the T1 pin to clock Timer 1 when in the counter mode. When 1, T1 acts as an output and toggles at every Timer 1 overflow.
TSTAT.1	T0RD	This bit is set, for Timer 0, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
TSTAT.0	T0OE	When 0, this bit allows the T0 pin to clock Timer 0 when in the counter mode. When 1, T0 acts as an output and toggles at every Timer 0 overflow.

SU00612A

Figure 5. Timer 0 And 1 Extended Status (TSTAT)

T2MOD Address:419
Bit Addressable
Reset Value: 00H

MSB LSB

—	—	RCLK1	TCLK1	—	T2RD	T2OE	DCEN

BIT	SYMBOL	FUNCTION
T2MOD.5	RCLK1	Receive Clock Flag.
T2MOD.4	TCLK1	Transmit Clock Flag. RCLK1 and TCLK1 are used to select Timer 2 overflow rate as a clock source for UART1.
T2MOD.2	T2RD	This bit is set for Timer 2, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
T2MOD.1	T2OE	When 0, this bit allows the T2 pin to clock Timer 2 when in the counter mode. When 1, T2 acts as an output and toggles at every Timer 2 overflow.
T2MOD.0	DCEN	Controls count direction for Timer 2 in autoreload mode. DCEN=1 counter set to count up DCEN=0 counter set to count down

SU00610

Figure 6. Timer 2 Mode Control (T2MOD)

WATCHDOG TIMER

The watchdog timer subsystem protects the system from incorrect code execution by causing a system reset when the watchdog timer underflows as a result of a failure of software to feed the timer prior to the timer reaching its terminal count.

Watchdog Function

The watchdog consists of a programmable prescaler and the main timer. The prescaler derives its clock from the on-chip oscillator. The prescaler consists of a programmable TCLK followed by a 13 stage counter with taps from stage 6 through stage 13. This is shown in Figure 7. The tap selection is also programmable. The watchdog main counter is a down counter clocked (decremented) each time the programmable prescaler underflows. The watchdog generates an underflow signal (and is autoloaded) when the watchdog is at count 0 and the clock to decrement the watchdog occurs. The watchdog is 8 bits long and the autoload value can range from 0 to FFH. (The autoload value of 0 is permissible since the prescaler is cleared upon autoload).

This leads to the following user design equations. Definitions :t_{OSC} is the oscillator period, N is the selected prescaler tap value, W is the main counter autoload value, t_{MIN} is the minimum watchdog time-out value (when the autoload value is 0), t_{MAX} is the maximum time-out value (when the autoload value is FFH), t_D is the design time-out value.

$t_{MIN} = t_{OSC} \times 4 \times 64$ (W = 0)

$t_{MAX} = t_{OSC} \times 64 \times 8192 \times 256$ (W = 255)

$t_D = t_{MIN} \times 2^{PRESCALER} \times (W + 1)$
(where prescaler = 0, 1, 2, 3, 4, 5, 6, or 7)

The watchdog timer is not directly loadable by the user. Instead, the value to be loaded into the main timer is held in an autoload register or is part of the mask ROM programming. In order to cause the main timer to be loaded with the appropriate value, a special sequence of software action must take place. This operation is referred to as feeding the watchdog timer.

To feed the watchdog, two instructions must be sequentially executed successfully. No intervening SFR accesses are allowed, so interrupts should be disabled before feeding the watchdog. The instructions should move A5H to the WFEED1 register and then 5AH to the WFEED2 register. If WFEED1 is correctly loaded and WFEED2 is not correctly loaded, then an immediate watchdog reset will occur.

The software must be written so that a feed operation takes place every t_D seconds from the last feed operation. Some tradeoffs may need to be made. It is not advisable to include feed operations in minor loops or in subroutines unless the feed operation is a specific subroutine.

Watchdog Control Register (WDCON) (Bit Addressable)

The following bits of this register are read only in the ROM part when $\overline{EA}$ is high: PRE0, PRE1, and PRE2. That is, the register will reflect the mask programmed values. In the ROM part with $\overline{EA}$ high, these bits are taken from mask coded bits and are not readable by the program.

The reset values of the WDCON and WDL registers will be such that the watchdog timer has a timeout period of $4 \times 64 \times t_{OSC}$. The watchdog timer will not generate an interrupt. WDCON can be written by software only by executing a valid watchdog feed sequence.

The watchdog timer subsystem consists of a programmable 13-bit prescaler, and an 8-bit main timer. The main timer is clocked by a tap taken from one of the top 8-bits of the prescaler. The clock source for the prescaler is the same as TCLK (same as the clock source for the timers). Thus the main counter can be clocked as often as once every 64* TCLKs (see Table 1).

Table 1. Prescaler Select Values in WDCON

PRE2	PRE1	PRE0	DIVISOR
0	0	0	TCLK*32*2
0	0	1	TCLK*32*4
0	1	0	TCLK*32*8
0	1	1	TCLK*32*16
1	0	0	TCLK*32*32
1	0	1	TCLK*32*64
1	1	0	TCLK*32*128
1	1	1	TCLK*32*256

NOTE:
Where, $t_{CLK} = t_{OSC} \times 4, \times 16, \times 64$ (set in SCR).

Programming the Watchdog Timer

Both the EPROM and ROM devices have a set of SFRs for holding the watchdog autoload values and the control bits. The watchdog time-out flag is present in the watchdog control register and operates the same in all versions. In the EPROM device, the watchdog parameters (autoload value and control) are always taken from the SFRs. In the ROM device, the watchdog parameters can be mask programmed or taken from the SFRs. The selection to take the watchdog parameters from the SFRs or from the mask programmed values is controlled by $\overline{EA}$ (external access). When $\overline{EA}$ is high (internal ROM access), the watchdog parameters are taken from the mask programmed values. When $\overline{EA}$ is low (external access), the watchdog parameters are taken from the SFRs. The user should be able to leave code in his program which initializes the watchdog SFRs even though he has migrated to the mask ROM part. This allows no code changes from EPROM prototyping to ROM coded production parts.

Watchdog Detailed Operation

EPROM Device (and ROMless Operation: $\overline{EA}$ = 0)

In the ROMless operation (ROM part, $\overline{EA}$ = 0) and in the EPROM device, the watchdog operates in the following manner.

When external $\overline{RESET}$ is applied, the following takes place:

- Watchdog run control bit set to ON.
- Autoload register WDL set to 00 (min. count).
- Watchdog time-out flag cleared.
- Prescaler is cleared.
- Prescaler tap set to the lowest divide.
- Autoload takes place.

Note that when coming out of a hardware reset, the software should load the autoload registers and then feed the watchdog (cause an autoload). The watchdog will now be starting at a known point.

If the watchdog is running and happens to underflow at the time the external RESET is applied, the watchdog time-out flag will be cleared.

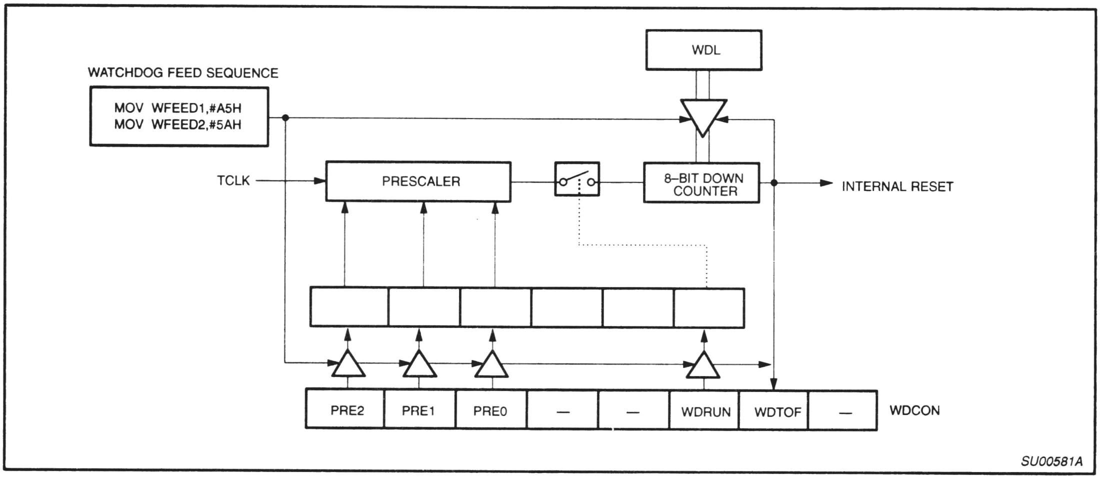


Figure 7. Watchdog Timer in XA-G1

When the watchdog underflows, the following action takes place (see Figure 7):

- Autoload takes place.
- Watchdog time-out flag is set
- Watchdog run bit unchanged.
- Autoload register unchanged.
- Prescaler tap unchanged.
- All other device action same as external reset.

Note that if the watchdog underflows, the program counter will be loaded from the reset vector as in the case of an internal reset. The watchdog time-out flag can be examined to determine if the watchdog has caused the reset condition. The watchdog time-out flag bit can be cleared by software.

WDCON Register Bit Definitions

WDCON.7	PRE2	Prescaler Select 2, reset to 1
WDCON.6	PRE1	Prescaler Select 1, reset to 1
WDCON.5	PRE0	Prescaler Select 0, reset to 1
WDCON.4	—	
WDCON.3	—	
WDCON.2	WDRUN, reset to 1	
WDCON.1	WDTOF	Timeout flag
WDCON.0	—	

UARTs

The XA-G1 includes 2 UART ports that are compatible with the enhanced UART used on the 8xC51FB. Baud rate selection is somewhat different due to the clocking scheme used for the XA timers.

Some other enhancements have been made to UART operation. The first is that there are separate interrupt vectors for each UART's transmit and receive functions. The second is double-buffering of the transmit register to allow time for interrupt processing without introducing inter-character gaps when tightly transmitted characters are required in the application. A break detect function has been added to the UART. This operates independently of the UART itself and provides a start-of-break status bit that the program may test. Finally, an Overrun Error flag has been added to detect missed characters in the received data stream.

Each UART rate is determined by either a fixed division of the oscillator (in UART modes 0 and 2) or by the timer 1 or timer 2 overflow rate (in UART modes 1 and 3).

The serial port receive and transmit registers are both accessed at Special Function Register SnBUF. Writing to SnBUF loads the transmit register, and reading SnBUF accesses a physically separate receive register.

The serial port can operate in 4 modes:

Mode 0: Serial I/O expansion mode. Serial data enters and exits through RxDn. TxDn outputs the shift clock. 8 bits are transmitted/received (LSB first). (The baud rate is fixed at 1/16 the oscillator frequency.)

Mode 1: Standard 8-bit UART mode. 10 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), and a stop bit (1). On receive, the stop bit goes into RB8 in Special Function Register SnCON. The baud rate is variable.

Mode 2: Fixed rate 9-bit UART mode. 11 bits are transmitted (through TxDn) or received (through RxDn): start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). On Transmit, the 9th data bit (TB8_n in SnCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8_n. On receive, the 9th data bit goes into RB8_n in Special Function Register SnCON, while the stop bit is ignored. The baud rate is programmable to 1/32 of the oscillator frequency.

Mode 3: Standard 9-bit UART mode. 11 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). In fact, Mode 3 is the same as Mode 2 in all respects except baud rate. The baud rate in Mode 3 is variable.

In all four modes, transmission is initiated by any instruction that uses SnBUF as a destination register. Reception is initiated in Mode 0 by the condition RI_n = 0 and REN_n = 1. Reception is initiated in the other modes by the incoming start bit if REN_n = 1.

Serial Port Control Register

The serial port control and status register is the Special Function Register SnCON, shown in Figure 9. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8_n and RB8_n), and the serial port interrupt bits (TI_n and RI_n).

CLOCKING SCHEME/BAUD RATE GENERATION

The XA UARTS clock rates are determined by either a fixed division (modes 0 and 2) of the oscillator clock or by the Timer 1 or Timer 2 overflow rate (modes 1 and 3).

The clock for the UARTs in XA runs at 16x the Baud rate. If the timers are used as the source for Baud Clock, since maximum speed of timers/Baud Clock is Osc/4, the maximum baud rate is timer overflow divided by 16 i.e. Osc/64.

In Mode 0, it is fixed at Osc/16. In Mode 2, however, the fixed rate is Osc/32.

Pre-scaler for all Timers T0,1,2 controlled by PT1, PT0 bits in SCR	00	Osc/4
	01	Osc/16
	10	Osc/64
	11	reserved

Baud Rate for UART Mode 0:

Baud_Rate=Osc/16

Baud Rate calculation for UART Mode 1 and 3:

Baud_Rate=Timer_Rate/16*N

Timer_Rate=Osc/(N*(Timer_Range– Timer_Reload_Value))

where N=the TCLK prescaler value: 4, 16, or 64.
and Timer_Range= 256 for timer 1 in mode 2.
65536 for timer 1 in mode 0 and timer 2 in count up mode.

The timer reload value may be calculated as follows:

Timer_Reload_Value=Timer_Range–(Osc/(Baud_Rate*N*16))

NOTES:

1. The maximum baud rate for a UART in mode 1 or 3 is Osc/64.
2. The lowest possible baud rate (for a given oscillator frequency and N value) may be found by using a timer reload value of 0.
3. The timer reload value may never be larger than the timer range.
4. If a timer reload value calculation gives a negative or fractional result, the baud rate requested is not possible at the given oscillator frequency and N value.

Baud Rate for UART Mode 2:

Baud_Rate = Osc/32

Using Timer 2 to Generate Baud Rates

Timer T2 is a 16-bit up/down counter in XA. As a baud rate generator, timer 2 is selected as a clock source for either/both UART0 and UART1 transmitters and/or receivers by setting TCLKn and/or RCLKn in T2CON and T2MOD. As the baud rate generator, T2 is incremented as Osc/N where N=4, 16 or 64 depending on TCLK as programmed in the SCR bits PT1, and PTO. So, if T2 is the source of one UART, the other UART could be clocked by either T1 overflow or fixed clock, and the UARTs could run independently with different baud rates.

T2CON 0x418		bit5	bit4	
		RCLK0	TCLK0	

T2MOD 0x419		bit5	bit4	
		RCLK1	TCLK1	

When Timer 1 or 2 is the source for baud clock, the baud rate is given by

Baud Rate=Osc/16*1/Timer Overflow Rate

The timer T2 or T1 (16-bit mode) reload value is set by

Up-counter Mode

reload=65,536–[Osc/N)*1/Baud Rate]
where N=4,16 or 64

Down-counter Mode

reload=[(Osc/16N)*1/Baud Rate]
where N=4, 16, or 64

Prescaler Select for Timer Clock (TCLK)

SCR 0x440		bit3	bit2	
		PT1	PT0	

SnSTAT Address: S0STAT 421
S1STAT 425
Bit Addressable
Reset Value: 00H

MSB LSB

—	—	—	—	FEn	BRn	OEn	STINTn

BIT	SYMBOL	FUNCTION
SnSTAT.3	FEn	Framing Error flag is set when the receiver fails to see a valid STOP bit at the end of the frame.
SnSTAT.2	BRn	Break Detect flag is set if a character is received with all bits (including STOP bit) being logic '0'. Thus it gives a "Start of Break Detect" on bit 8 for Mode 1 and bit 9 for Modes 2 and 3. The break detect feature operates independently of the UARTs and provides the START of Break Detect status bit that a user program may poll.
SnSTAT.1	OEn	Overrun Error flag is set if a new character is received in the receiver buffer while it is still full (before the software has read the previous character from the buffer), i.e., when bit 8 of a new byte is received while RI in SnCON is still set.
SnSTAT.0	STINTn	This flag must be set to enable any of the above status flags to generate a receive interrupt (RIn). The only way it can be cleared is by a software write to this register.

SU00607A

Figure 8. Serial Port Extended Status (SnSTAT) Register
(See also Figure 10 regarding Framing Error flag.)

INTERRUPT SCHEME

There are separate interrupt vectors for each UART's transmit and receive functions.

Table 2. Vector Locations for UARTs in XA

Vector Address	Interrupt Source	Arbitration
9CH – 9FH	Uart 1 Receiver	8
A0H – A3H	Uart 1 Transmitter	9
A4H – A7H	Uart 2 Receiver	10
A8H – ABH	Uart 2 Transmitter	11

NOTE:
The transmit and receive vectors could contain the same ISR address to work like a 8051 interrupt scheme

Error Handling, Status Flags and Break Detect
The UARTs in XA has the following error flags; see Figure 8.

Multiprocessor Communications

Modes 2 and 3 have a special provision for multiprocessor communications. In these modes, 9 data bits are received. The 9th one goes into RB8. Then comes a stop bit. The port can be programmed such that when the stop bit is received, the serial port interrupt will be activated only if RB8 = 1. This feature is enabled by setting bit SM2 in SCON. A way to use this feature in multiprocessor systems is as follows:

When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte differs from a data byte in that the 9th bit is 1 in an address byte and 0 in a data byte. With SM2 = 1, no slave will be interrupted by a data byte. An address byte, however, will interrupt all slaves, so that each slave can examine the received byte and see if it is being addressed. The addressed slave will clear its SM2 bit and prepare to receive the data bytes that will be coming. The slaves that weren't being addressed leave their SM2s set and go on about their business, ignoring the coming data bytes.

SM2 has no effect in Mode 0, and in Mode 1 can be used to check the validity of the stop bit although this is better done with the Framing Error (FE) flag. In a Mode 1 reception, if SM2 = 1, the receive interrupt will not be activated unless a valid stop bit is received.

Automatic Address Recognition
Automatic Address Recognition is a feature which allows the UART to recognize certain addresses in the serial bit stream by using hardware to make the comparisons. This feature saves a great deal of software overhead by eliminating the need for the software to examine every serial address which passes by the serial port. This feature is enabled by setting the SM2 bit in SCON. In the 9 bit UART modes, mode 2 and mode 3, the Receive Interrupt flag (RI) will be automatically set when the received byte contains either the "Given" address or the "Broadcast" address. The 9 bit mode requires that the 9th information bit is a 1 to indicate that the received information is an address and not data. Automatic address recognition is shown in Figure 11.

Using the Automatic Address Recognition feature allows a master to selectively communicate with one or more slaves by invoking the Given slave address or addresses. All of the slaves may be contacted by using the Broadcast address. Two special Function Registers are used to define the slave's address, SADDR, and the address mask, SADEN. SADEN is used to define which bits in the SADDR are to be used and which bits are "don't care". The SADEN mask can be logically ANDed with the SADDR to create the "Given" address which the master will use for addressing each of the slaves. Use of the Given address allows multiple slaves to be recognized while excluding others. The following examples will help to show the versatility of this scheme:

Slave 0 SADDR = 1100 0000
SADEN = 1111 1101
Given = 1100 00X0

Slave 1 SADDR = 1100 0000
SADEN = 1111 1110
Given = 1100 000X

In the above example SADDR is the same and the SADEN data is used to differentiate between the two slaves. Slave 0 requires a 0 in bit 0 and it ignores bit 1. Slave 1 requires a 0 in bit 1 and bit 0 is ignored. A unique address for Slave 0 would be 1100 0010 since slave 1 requires a 0 in bit 1. A unique address for slave 1 would be 1100 0001 since a 1 in bit 0 will exclude slave 0. Both slaves can be selected at the same time by an address which has bit 0 = 0 (for slave 0) and bit 1 = 0 (for slave 1). Thus, both could be addressed with 1100 0000.

In a more complex system the following could be used to select slaves 1 and 2 while excluding slave 0:

Slave 0 SADDR = 1100 0000
SADEN = 1111 1001
Given = 1100 0XX0

Slave 1 SADDR = 1110 0000
SADEN = 1111 1010
Given = 1110 0X0X

Slave 2 SADDR = 1110 0000
SADEN = 1111 1100
Given = 1110 00XX

In the above example the differentiation among the 3 slaves is in the lower 3 address bits. Slave 0 requires that bit 0 = 0 and it can be uniquely addressed by 1110 0110. Slave 1 requires that bit 1 = 0 and it can be uniquely addressed by 1110 and 0101. Slave 2 requires that bit 2 = 0 and its unique address is 1110 0011. To select Slaves 0 and 1 and exclude Slave 2 use address 1110 0100, since it is necessary to make bit 2 = 1 to exclude slave 2.

The Broadcast Address for each slave is created by taking the logical OR of SADDR and SADEN. Zeros in this result are teated as don't-cares. In most cases, interpreting the don't-cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR and SADEN are loaded with 0s. This produces a given address of all "don't cares" as well as a Broadcast address of all "don't cares". This effectively disables the Automatic Addressing mode and allows the microcontroller to use standard UART drivers which do not make use of this feature.

SnCON Address: S0CON 420
S1CON 424

Bit Addressable
Reset Value: 00H

MSB							LSB
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Where SM0, SM1 specify the serial port mode, as follows:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	shift register	fosc/ 16
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	fosc/32
1	1	3	9-bit UART	variable

BIT	SYMBOL	FUNCTION
SnCON.5	SM2	Enables the multiprocessor communication feature in Modes 2 and 3. In Mode 2 or 3, if SM2 is set to 1, then RI will not be activated if the received 9th data bit (RB8) is 0. In Mode 1, if SM2=1 then RI will not be activated if a valid stop bit was not received. In Mode 0, SM2 should be 0.
SnCON.4	REN	Enables serial reception. Set by software to enable reception. Clear by software to disable reception.
SnCON.3	TB8	The 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired.
SnCON.2	RB8	In Modes 2 and 3, is the 9th data bit that was received. In Mode 1, it SM2=0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
SnCON.1	TI	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
SnCON.0	RI	Receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

SU00597A

Figure 9. Serial Port Control (SnCON) Register

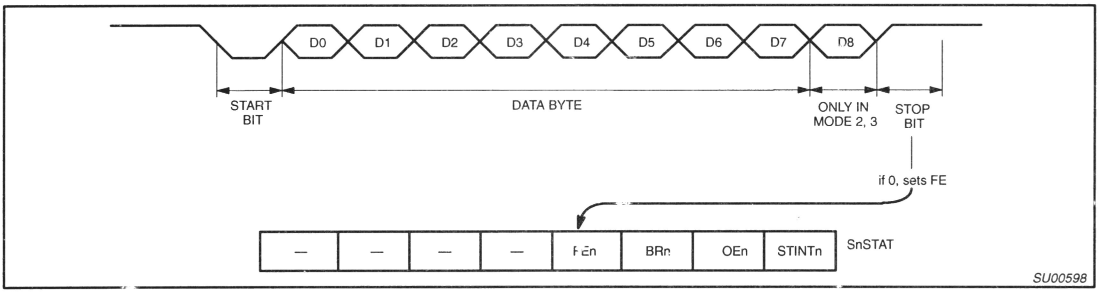
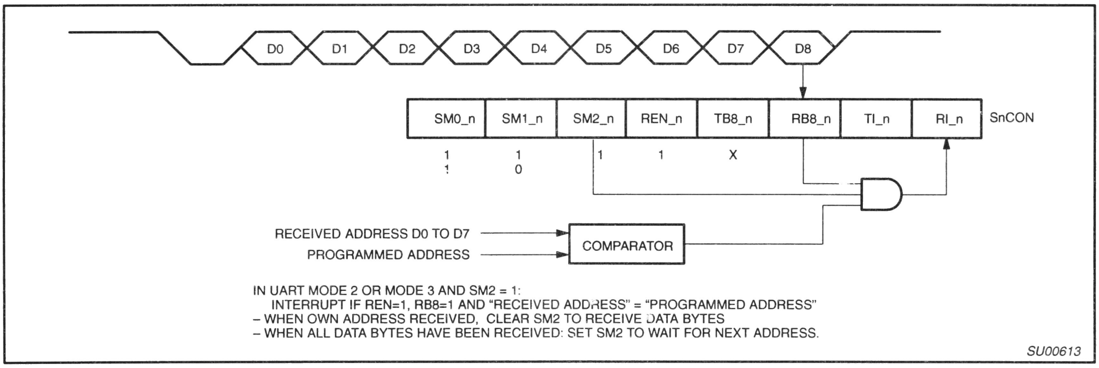

D0
D1
D2
D3
D4
D5
D6
D7
D8
START BIT
DATA BYTE
ONLY IN MODE 2, 3
STOP BIT
if 0, sets FE
—
—
—
—
FEn
BRn
OEn
STINTn
SnSTAT
SU00598

Figure 10. UART Framing Error Detection


D0
D1
D2
D3
D4
D5
D6
D7
D8
SM0_n
SM1_n
SM2_n
REN_n
TB8_n
RB8_n
TI_n
RI_n
SnCON
1
1
1
0
1
1
1
X
RECEIVED ADDRESS D0 TO D7
PROGRAMMED ADDRESS
COMPARATOR

IN UART MODE 2 OR MODE 3 AND SM2 = 1:
INTERRUPT IF REN=1, RB8=1 AND "RECEIVED ADDRESS" = "PROGRAMMED ADDRESS"
– WHEN OWN ADDRESS RECEIVED, CLEAR SM2 TO RECEIVE DATA BYTES
– WHEN ALL DATA BYTES HAVE BEEN RECEIVED: SET SM2 TO WAIT FOR NEXT ADDRESS.

SU00613

Figure 11. UART Multiprocessor Communication, Automatic Address Recognition

I/O PORT OUTPUT CONFIGURATION

Each I/O port pin on the XA-G1 can be user configured to one of 4 output types. The types are Quasi-bidirectional (essentially the same as standard 80C51 family I/O ports), Open-Drain, Push-Pull, and Off (high impedance). The default configuration after reset is Quasi-bidirectional. However, in the ROMless mode (the $\overline{EA}$ pin is low at reset), the port pins that comprise the external data bus will default to push-pull outputs.

I/O port output configurations are determined by the settings in port configuration SFRs. There are 2 SFRs for each port, called PnCFGA and PnCFGB, where "n" is the port number. One bit in each of the 2 SFRs relates to the output setting for the corresponding port pin, allowing any combination of the 2 output types to be mixed on those port pins. For instance, the output type of port 1 pin 3 is controlled by the setting of bit 3 in the SFRs P1CFGA and P1CFGB.

Table 3 shows the configuration register settings for the 4 port output types. The electrical characteristics of each output type may be found in the DC Characteristic table.

Table 3. Port Configuration Register Settings

PnCFGB	PnCFGA	Port Output Mode
0	0	Open Drain
0	1	Quasi-bidirectional
1	0	Off (high impedance)
1	1	Push-Pull

NOTE:
Mode changes may cause glitches to occur during transitions. When modifying both registers, WRITE instructions should be carried out consecutively.

EXTERNAL BUS

The external program/data bus on the XA-G1 allows for 8-bit or 16-bit bus width, and address sizes from 12 to 20 bits. The bus width is selected by an input at reset (see Reset Options below), while the address size is set by the program in a configuration register. If all off-chip code is selected (through the use of the $\overline{EA}$ pin), the initial code fetches will be done with the maximum address size (20 bits).

RESET

The device is reset whenever a logic "0" is applied to $\overline{RST}$ for at least 10 microseconds, placing a low level on the pin re-initializes the on-chip logic.

The duration of reset must be extended when power is initially applied or when using reset to exit power down mode. This is due to the need to allow the oscillator time to start up and stabilize. For most power supply ramp up conditions, this time is 10 milliseconds.

As it is brought high again, an exception is generated which causes the processor to jump to the address contained in the memory location 0000. The destination of the reset jump must be located in the first 64k of code address on power-up, all vectors are 16-bit values and so point to page zero addresses only. After a reset the RAM contents are indeterminate.

RESET OPTIONS

The $\overline{EA}$ pin is sampled on the rising edge of the $\overline{RST}$ pulse, and determines whether the device is to begin execution from internal or external code memory. $\overline{EA}$ pulled high configures the XA in single-chip mode. If $\overline{EA}$ is driven low, the device enters ROMless mode. After Reset is released, the $\overline{EA}$/WAIT pin becomes a bus wait signal for external bus transactions.

The BUSW/P3.5 pin is weakly pulled high while reset is asserted, allowing simple biasing of the pin with a resistor to ground to select the alternate bus width.

POWER REDUCTION MODES

The XA-G1 supports Idle and Power down modes of power reduction. The idle mode leaves some peripherals running to allow them to wake up the processor when an interrupt is generated. The power down mode stops the processor clock in order to absolutely minimize power. The processor can be made to exit power down mode via reset or one of the external interrupt inputs. In power down mode, the power supply voltage may be further reduced to the keep-alive voltage, retaining the RAM, register, and SFR values at the point where the power down mode was entered.

INTERRUPTS

The XA-G1 supports 31 maskable interrupts vectored interrupt sources. The maskable interrupts each have 16 priority levels and may be globally and/or individually enabled or disabled.

The XA defines four types of interrupts:

- **Exception Interrupts** – These are system level errors and other very important occurrences which include stack overflow, divide-by-0, and reset.
- **Event interrupts** – These are peripheral interrupts from devices such as UARTs, timers, and external interrupt inputs.
- **Software Interrupts** – These are equivalent of hardware interrupt, but are requested only under software control.
- **Trap Interrupts** – These are TRAP instructions, generally used to call system services in a multi-tasking system.

Exception interrupts, software interrupts, and trap interrupts are generally standard for XA derivatives and are detailed in the XA User Guide. Event interrupts tend to be different on different XA derivatives.

The XA-G1 supports a total of 9 maskable event interrupt sources (for the various XA-G1 peripherals), seven software interrupts, 5 exception interrupts (plus reset), and 16 traps. The maskable event interrupts share a global interrupt enable bit (the EA bit in the IEL register) and each also has a separate individual interrupt enable bit (in the IEL or IEH registers). Each event interrupt can be set to occur at one of 8 priority levels (levels 8 through 15) via bits in the Interrupt Priority (IP) registers, IPA0 through IPA5. Details of the priority scheme may be found in the XA User Guide.

The complete interrupt vector list for the XA-G1, including all 4 interrupt types, is shown in the following tables. The tables include the address of the vector for each interrupt, the related priority register bits (if any), and the arbitration ranking for that interrupt source. The arbitration ranking determines the order in which interrupts are processed if more than one interrupt of the same priority occurs simultaneously.

Table 4. Interrupt Vectors

EXCEPTION/TRAPS PRECEDENCE

DESCRIPTION	VECTOR ADDRESS	ARBITRATION RANKING
Reset (h/w, watchdog, s/w)	0000–0003	0 (High)
Breakpoint (h/w trap 1)	0004–0007	1
Trace (h/w trap 2)	0008–000B	1
Stack Overflow (h/w trap 3)	000C–000F	1
Divide by 0 (h/w trap 4)	0010–0013	1
User RETI (h/w trap 5)	0014–0017	1
TRAP 0– 15 (software)	0040–007F	1

EVENT INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY	ARBITRATION RANKING
External interrupt 0	IE0	0080–0083	EX0	IPA0.3–0	2
Timer 0 interrupt	TF0	0084–0087	ET0	IPA0.7–4	3
External interrupt 1	IE1	0088–008B	EX1	IPA1.3–0	4
Timer 1 interrupt	TF1	008C–008F	ET1	IPA1.7–4	5
Timer 2 interrupt	TF2	0090–0093	ET2	IPA2.3–0	6
Serial port 0 Rx	RI.0	00A0–00A3	ERI0	IPA4.3–0	7
Serial port 0 Tx	TI.0	00A4–00A7	ETI0	IPA4.7–4	8
Serial port 1 Rx	RI.1	00A8–00AB	ERI1	IPA5.3–0	9
Serial port 1 Tx	TI.1	00AC–00AF	ETI1	IPA5.7–4	10

SOFTWARE INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY
Software interrupt 1	SWR1	0100–0103	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010B	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010F	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0018–011B	SWE7	(fixed at 7)

ABSOLUTE MAXIMUM RATINGS

PARAMETER	RATING	UNIT
Operating temperature under bias	–55 to +125	°C
Storage temperature range	–65 to +150	°C
Voltage on $\overline{EA}$/V_{PP} pin to V_{SS}	0 to +13.0	V
Voltage on any other pin to V_{SS}	–0.5 to V_{DD}+0.5V	V
Maximum I_{OL} per I/O pin	15	mA
Power dissipation (based on package heat transfer limitations, not device power consumption)	1.5	W

DC ELECTRICAL CHARACTERISTICS

V_{DD} = 5.0V ±10% to 3.0V ±10% unless otherwise specified;
T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial, unless otherwise specified

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS			UNIT
			MIN	TYP	MAX	
Supplies						
I_{DD}	Supply current operating	5.0V, 30 MHz			100	mA
I_{ID}	Idle mode supply current	5.0V, 30 MHz			25	mA
I_{PD}	Power-down current	5.0V, 3.0V		5	50	µA
V_{RAM}	RAM-keep-alive voltage	Ram-keep-alive voltage	1.5			V
V_{IL}	Input low voltage, except		–0.5		0.8	V
V_{IH}	Input high voltage, except XTAL1, RST	At 5.0V[1]	2.2			V
		At 3.0V[1]	2			V
V_{IH1}	Input high voltage to XTAL1, RST	For both 3.0 & 5.0V	0.7V_{DD}			V
V_{OL}	Output low voltage all ports, ALE, PSEN[5]	I_{OL} = 3.2mA, V_{DD} = 5.0V			0.8	V
		1.0mA, V_{DD} = 3.0V				V
V_{OH1}	Output high voltage all ports, ALE, PSEN[3]	I_{OH} = –100µA, V_{DD} = 5.0V	2.4			V
		I_{OH} = –30µA, V_{DD} = 3.0V	2.2			V
V_{OH2}	Output high voltage, ports P0–3, ALE, PSEN[4]	I_{OH} = 3.2mA, V_{DD} = 5.0V	2.4			V
		I_{OH} = 1mA, V_{DD} = 3.0V	2.2			V
C_{IO}	Input/Output pin capacitance[2]				15	pF
I_{IL}	Logical 0 input current, P0–3[8]	V_{IN} = 0.45V			–50	µA
I_{LI}	Input leakage current, P0–3[7]				±10	µA
I_{TL}	Logical 1 to 0 transition current all ports[6]	At 6V			–650	µA
		At 3V			–250	µA

NOTE:

1. Values are linear in between
2. Max. 15pF for~EA/V_{PP}
3. Ports in Quasi bi-directional mode with weak pull-up
4. Ports in Push-Pull mode, both pull-up and pull-down assumed to be same strength
5. In all output modes
6. Port pins source a transition current when used in quasi-bidirectional mode and externally driven from 1 to 0. This current is highest when V_{IN} is approximately 2V.
7. Measured with port in high impedance output mode.
8. Measured with port in quasi-bidirectional output mode.
9. Load capacitance for port 0, ALE, and $\overline{PSEN}$=100pF, load capacitance for all other outputs = 80pF.
10. Under steady state (non-transient) conditions, I_{OL} must be extremely limited as follows:

    Maximum I_{OL} per port pin: 15mA (*NOTE: This is 85°C specification.)
    Maximum I_{OL} per 8-bit port: 26mA
    Maximum total I_{OL} for all output: 71mA

    If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.

AC ELECTRICAL CHARACTERISTICS

V_{DD} = 5.0V ±10% or 3.0V ±10%, T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial.

SYMBOL	PARAMETER	30MHz[1]			16MHz[2]			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
External Clock								
f_C	Oscillator frequency		30			16		MHz
$t_C = 1/f_C$	Clock period and CPU timing cycle		$1/f_C$			$1/f_C$		ns
t_{CHCX}	Clock high-time (60%–40% duty cycle)		$(t_C/2) * 0.4$			$(t_C/2) * 0.4$		ns
t_{CLCX}	Clock low-time (60%–40% duty cycle)		$(t_C/2) * 0.4$			$(t_C/2) * 0.4$		ns
t_{CLCH}	Clock rise-time		5			10		ns
t_{CHCL}	Clock fall-time		5			10		ns
Address Cycle								
t_{CRAR}	Delay from clock rising edge to ALE rising edge		24			38		ns
t_{LHLL}	ALE pulse width (programmable)		$(N+0.5) * t_C$			$(N+0.5) * t_C$		ns
t_{AVLL}	Address valid to ALE de-asserted (set-up)		t_{LLHL}			$t_{LLHL} - 4$		ns
t_{LLAX}	Address hold after ALE de-asserted		12			30		ns
Code Read Cycle								
t_{LLPL}	ALE de-asserted to PSEN active		$(t_C/2) + 10$			$(t_C/2) + 1$		ns
t_{AVIV}	Address valid to instruction valid (access time)		$(M*t_C) - 16$			$(M*t_C) - 21$		ns
t_{PLIV}	$\overline{\text{PSEN}}$ low to instruction valid		$(O*t_C) - 16$			$(O*t_C) - 21$		ns
t_{PLPH}	$\overline{\text{PSEN}}$ pulse width		$(O*t_C) - 5$			$O*t_C$		ns
t_{PXIX}	Instruction hold after $\overline{\text{PSEN}}$ de-asserted	0			0			ns
t_{PXIZ}	Bus 3-State after $\overline{\text{PSEN}}$ de-asserted		29			54		ns
t_{UAPH}	Hold time of unlatched port of address after $\overline{\text{PSEN}}$ is de-asserted.	0			0			ns
Data Read Cycle								
t_{RLRH}	$\overline{\text{RD}}$ pulse width		$(P*t_C) - 8$			$P*t_C$		ns
t_{LLRL}	ALE falling edge to $\overline{\text{RD}}$ falling edge		$(t_C/2) + 9$			$(t_C/2) + 1$		ns
t_{AVDV}	Data input valid after address valid (access time)		$(P*t_C) - 16$			$(P*t_C) - 21$		ns
t_{RLDV}	$\overline{\text{RD}}$ low to valid data in, enable time		$(P*t_C) - 16$			$(P*t_C) - 21$		ns
t_{RHDX}	Data hold time after $\overline{\text{RD}}$ de-asserted	0			0			ns
t_{RHDZ}	Bus 3-State after $\overline{\text{RD}}$ de-asserted		29			54		ns
t_{UARH}	Hold time of unlatched port of address after $\overline{\text{RD}}$ is de-asserted.	0			0			ns
Data Write Cycle								
t_{WLWH}	$\overline{\text{WR}}$ pulse width		$Q*t_C$			$Q*t_C$		ns
t_{LLWL}	ALE falling edge to $\overline{\text{WR}}$ asserted		$(t_C/2) + 5$			$(t_C/2) - 5$		ns
t_{QVWX}	Data valid before $\overline{\text{WR}}$ active (setup time)		$(R*t_C) - 7$			$(R*t_C) - 12$		ns
t_{WHQX}	Data hold time after $\overline{\text{WR}}$ rising edge		0			4		ns
t_{AVWL}	address valid to $\overline{\text{WR}}$ active		$(R*t_C) - 3$			$(R*t_C) - 8$		ns
t_{UAWH}	Hold time of unlatched part of address after $\overline{\text{WR}}$ is de-asserted		0			3		ns
WAIT Input								
t_{WTV}	Rising edge of Wait after $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{PSEN}}$ falling edge		$(S*t_C) - 15$			$(S*t_C) - 14$		ns
t_{WAIT}	CPU wait state period		t_C			t_C		ns

SYMBOL	PARAMETER	30MHz[1]			16MHz[2]			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
Input Pulse Width								
t_{PWI}	Pulse width for interrupt inputs		2*t_C			2*t_C		ns
t_{PWT}	Pulse width for timer inputs		2*t_C			2*t_C		ns
Shift Register								
t_{XLXL}	Serial port clock cycle time		16*t_C			16*t_C		ns
t_{QVXH}	Output data setup to clock rising edge		(2*t_C) – 30			(2*t_C) – 40		ns
t_{XHQX}	Output data hold to clock rising edge		(2*t_C) – 30			(2*t_C) – 40		ns
t_{XHDX}	Input data hold after clock rising edge		0			0		ns
t_{XHDV}	Clock rising edge to input data valid		30			40		ns

NOTES:
1. All values indicated for V_{DD} = 5V ±10%. Typical values are for 20°C.
2. All values indicated for V_{DD} = 3V ±10%. Typical values are for 20°C.
3. N = ALEW bit value
   M = burst mode code read clocks
   O = PSEN clocks
   P = RD clocks
   Q = WR clocks
   R = WR setup clocks

EXPLANATION OF THE AC SYMBOLS

Each timing symbol has five characters. The first character is always ‘t’ (= time). The other characters, depending on their positions, indicate the name of a signal or the logical status of that signal. The designations are:

A – Address
C – Clock
D – Input data
H – Logic level high
I – Instruction (program memory contents)
L – Logic level low, or ALE
P – $\overline{\text{PSEN}}$
Q – Output data
R – $\overline{\text{RD}}$ signal
t – Time
U – Undefined
V – Valid
W – $\overline{\text{WR}}$ signal
X – No longer a valid logic level
Z – Float

Examples: t_{AVLL} = Time for address valid to ALE low.
t_{LLPL} =Time for ALE low to $\overline{\text{PSEN}}$ low.

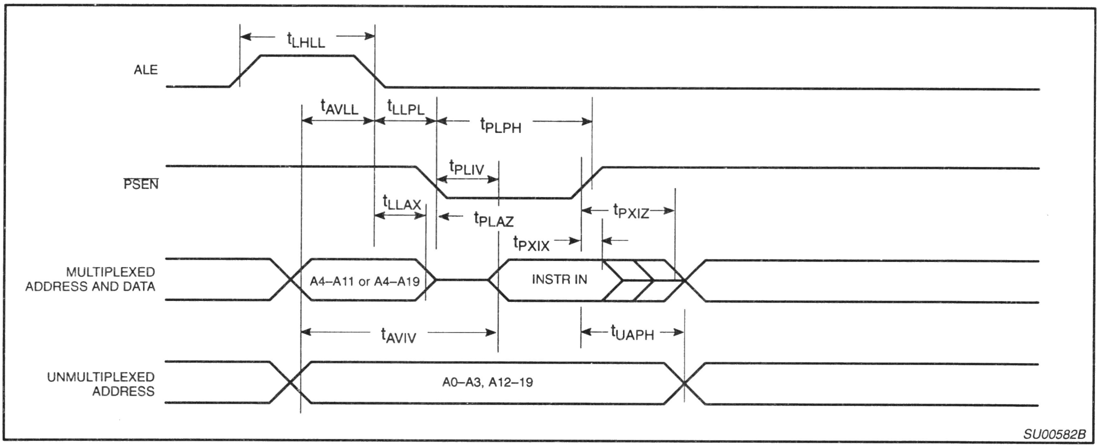

t_{LHLL}
ALE
t_{AVLL}
t_{LLPL}
t_{PLPH}
t_{PLIV}
$\overline{\text{PSEN}}$
t_{LLAX}
t_{PLAZ}
t_{PXIZ}
t_{PXIX}
MULTIPLEXED ADDRESS AND DATA
A4–A11 or A4–A19
INSTR IN
t_{AVIV}
t_{UAPH}
UNMULTIPLEXED ADDRESS
A0–A3, A12–19
SU00582B

Figure 12. External Program Memory Read Cycle

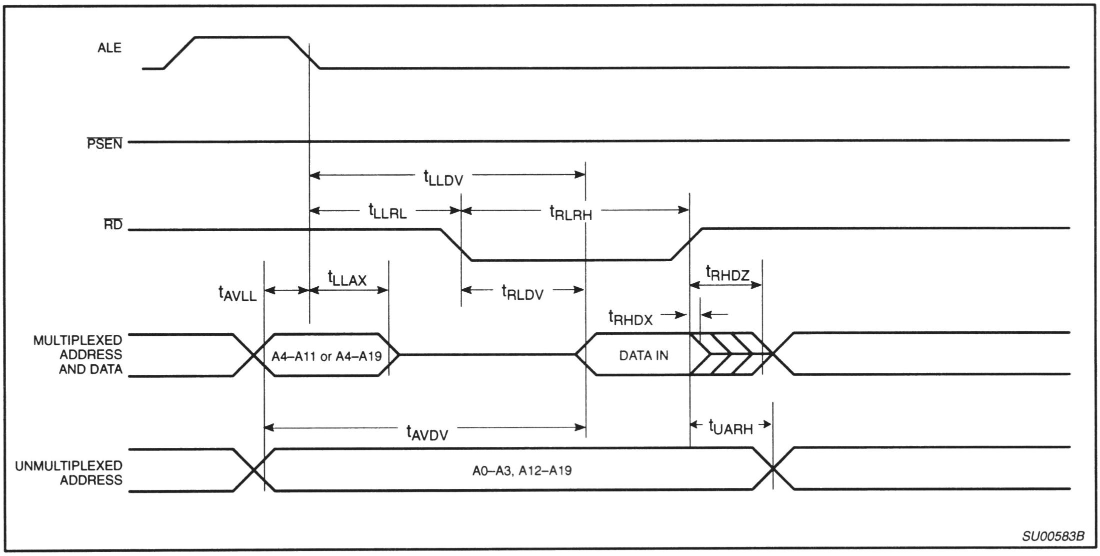

ALE
PSEN
RD
MULTIPLEXED ADDRESS AND DATA
UNMULTIPLEXED ADDRESS
tLLDV
tLLRL
tRLRH
tAVLL
tLLAX
tRLDV
tRHDZ
tRHDX
A4–A11 or A4–A19
DATA IN
tAVDV
tUARH
A0–A3, A12–A19
SU00583B

Figure 13. External Data Memory Read Cycle

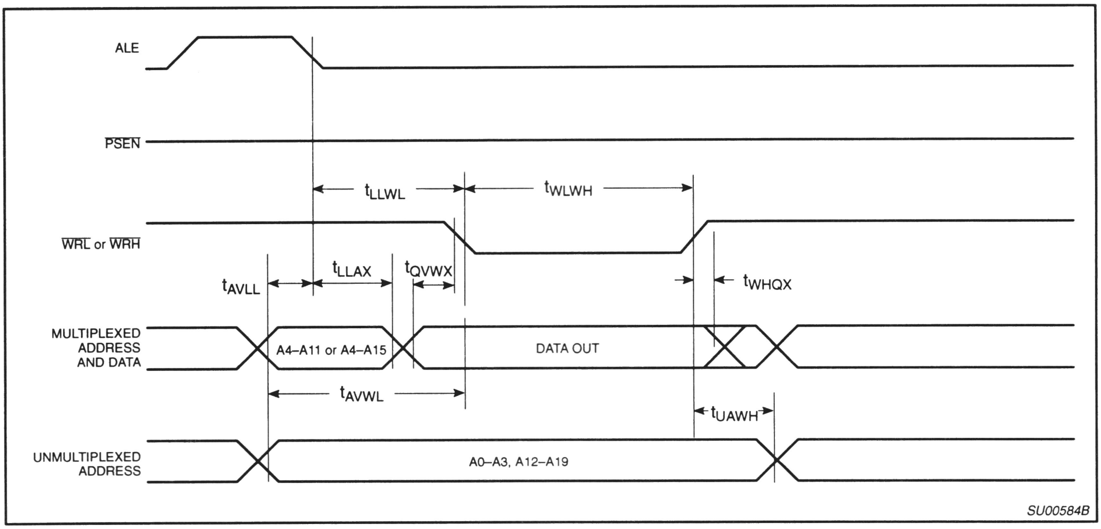

ALE
PSEN
WRL or WRH
MULTIPLEXED ADDRESS AND DATA
UNMULTIPLEXED ADDRESS
tLLWL
tWLWH
tAVLL
tLLAX
tQVWX
tWHQX
A4–A11 or A4–A15
DATA OUT
tAVWL
tUAWH
A0–A3, A12–A19
SU00584B

Figure 14. External Data Memory Write Cycle

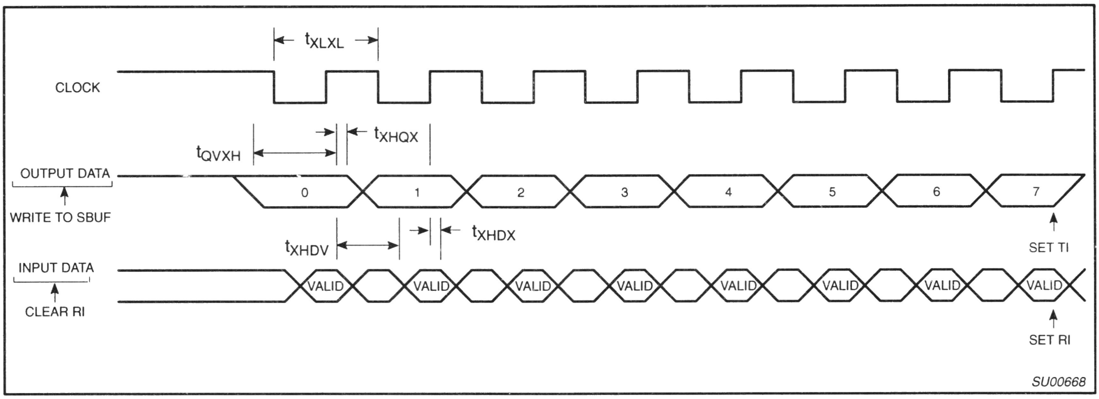

t_{XLXL}
CLOCK
t_{QVXH}
t_{XHQX}
OUTPUT DATA
WRITE TO SBUF
0
1
2
3
4
5
6
7
SET TI
t_{XHDV}
t_{XHDX}
INPUT DATA
CLEAR RI
VALID
VALID
VALID
VALID
VALID
VALID
VALID
VALID
SET RI
SU00668

Figure 15. Shift Register Mode Timing

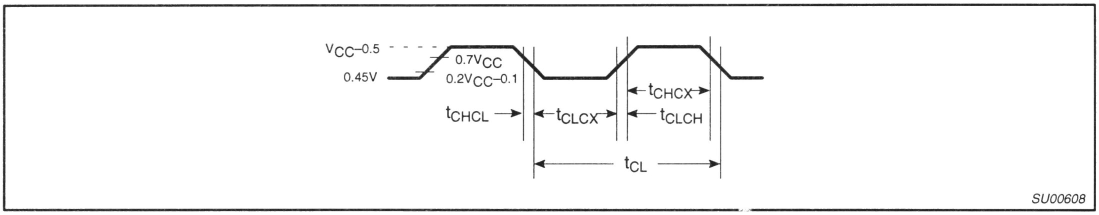
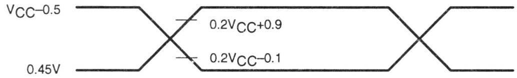

V_{CC}–0.5
0.45V
$0.7V_{CC}$
$0.2V_{CC}$–0.1
t_{CHCL}
t_{CLCX}
t_{CHCX}
t_{CLCH}
t_{CL}
SU00608

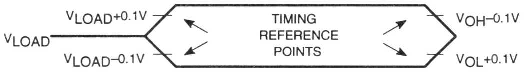
Figure 16. External Clock Drive


V_{CC}–0.5
0.45V
$0.2V_{CC}$+0.9
$0.2V_{CC}$–0.1

NOTE:
AC inputs during testing are driven at V_{CC} –0.5 for a logic '1' and 0.45V for a logic '0'.
Timing measurements are made at V_{IH} min for a logic '1' and V_{IL} max for a logic '0'.

SU00010

Figure 17. AC Testing Input/Output


V_{LOAD}+0.1V
V_{LOAD}
V_{LOAD}–0.1V
TIMING
REFERENCE
POINTS
V_{OH}–0.1V
V_{OL}+0.1V

NOTE:
For timing purposes, a port is no longer floating when a 100mV change from load voltage occurs, and begins to float when a 100mV change from the loaded V_{OH}/V_{OL} level occurs. $I_{OH}/I_{OL} \geq \pm 20mA$.

SU00011

Figure 18. Float Waveform

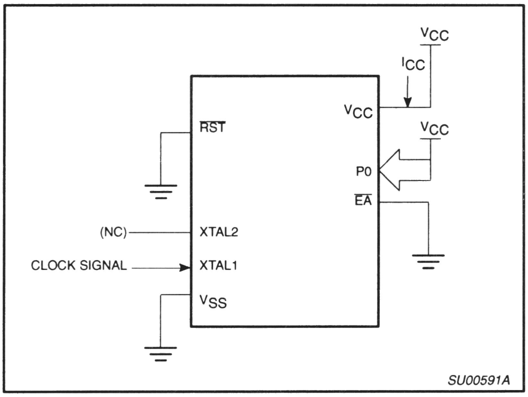

VCC
ICC
VCC
RST
VCC
P0
EA
(NC)
XTAL2
CLOCK SIGNAL
XTAL1
VSS
SU00591A

Figure 19. I_{CC} **Test Condition, Active Mode**
All other pins are disconnected

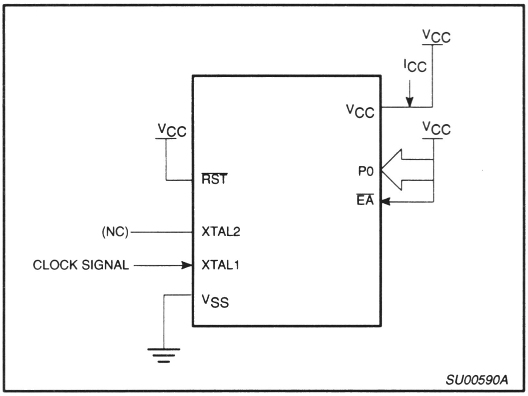

VCC
ICC
VCC
VCC
VCC
RST
P0
EA
(NC)
XTAL2
CLOCK SIGNAL
XTAL1
VSS
SU00590A

Figure 20. I_{CC} **Test Condition, Idle Mode**
All other pins are disconnected

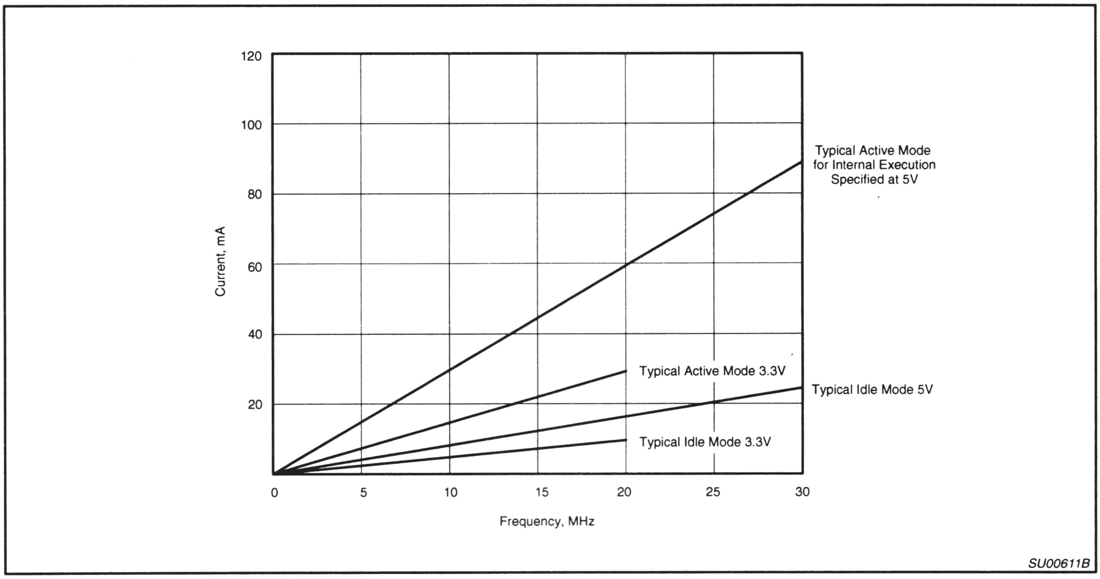


Figure 21. I_{CC} **vs. Frequency**
Valid only within frequency specification of the device under test.

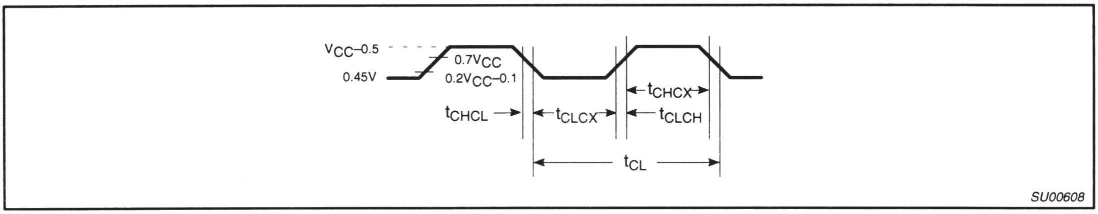

VCC–0.5
0.45V
0.7VCC
0.2VCC–0.1
tCHCL
tCLCX
tCHCX
tCLCH
tCL
SU00608

Figure 22. Clock Signal Waveform for I_{CC} **Tests in Active and Idle Modes**
$t_{CLCH} = t_{CHCL} = 5ns$

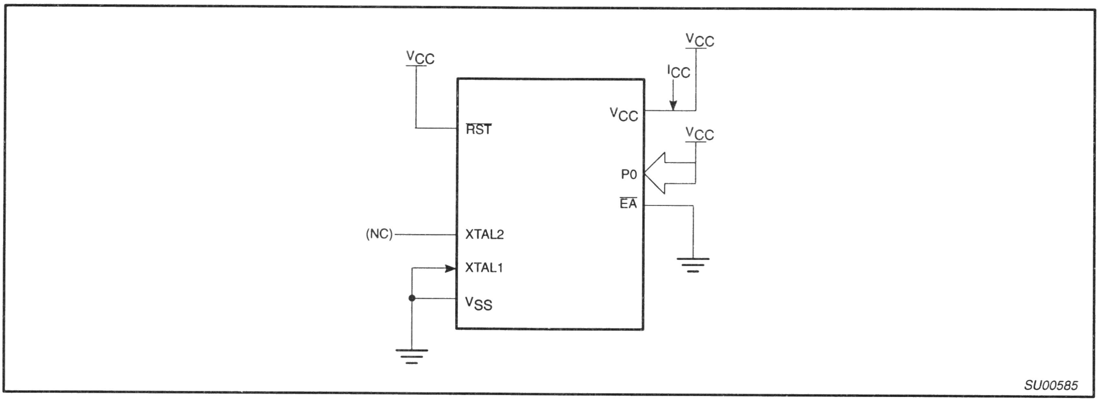


Figure 23. I_{CC} Test Condition, Power Down Mode
All other pins are disconnected. V_{CC}=2V to 5.5V

EPROM CHARACTERISTICS

The XA-G1 is programmed by using a modified Improved Quick-Pulse Programming™ algorithm. This algorithm is essentially the same as that used by the later 80C51 family EPROM parts. However different pins are used for many programming functions.

The XA-G1 contains three signature bytes that can be read and used by an EPROM programming system to identify the device. The signature bytes identify the device as an XA-G1 manufactured by Philips.

Table 5 shows the logic levels for reading the signature byte, and for programming the code memory and the security bits. The circuit configuration and waveforms for quick-pulse programming are shown in Figure 24. Figure 26 shows the circuit configuration for normal code memory verification.

Quick-Pulse Programming

The setup for microcontroller quick-pulse programming is shown in Figure 24. Note that the XA-G1 is running with a 3.5 to 12MHz oscillator. The reason the oscillator needs to be running is that the device is executing internal address and program data transfers.

The address of the EPROM location to be programmed is applied to ports 2 and 3, as shown in Figure 24. The code byte to be programmed into that location is applied to port 0. RST, $\overline{\text{PSEN}}$ and pins of port 1 specified in Table 5 are held at the 'Program Code Data' levels indicated in Table 5. The ALE/$\overline{\text{PROG}}$ is pulsed low 5 times as shown in Figure 25.

To program the security bits, repeat the 5 pulse programming sequence using the 'Pgm Security Bit' levels. After one security bit is programmed, further programming of the code memory and encryption table is disabled. However, the other security bits can still be programmed.

Note that the $\overline{\text{EA}}/V_{PP}$ pin must not be allowed to go above the maximum specified V_{PP} level for any amount of time. Even a narrow glitch above that voltage can cause permanent damage to the device. The V_{PP} source should be well regulated and free of glitches and overshoot.

Program Verification

If security bits 2 and 3 have not been programmed, the on-chip program memory can be read out for program verification. The address of the program memory locations to be read is applied to ports 2 and 3 as shown in Figure 26. The other pins are held at the 'Verify Code Data' levels indicated in Table 5. The contents of the address location will be emitted on port 0.

Reading the Signature Bytes

The signature bytes are read by the same procedure as a normal verification of locations 030H, 031H, and 060H except that P1.2 and P1.3 need to be pulled to a logic low. The values are:

(030H) = 15H indicates manufactured by Philips
(031H) = EAH indicates XA architecture
(060H) = 03H indicates XA-G1

Program/Verify Algorithms

Any algorithm in agreement with the conditions listed in Table 5, and which satisfies the timing specifications, is suitable.

Erasure Characteristics

Erasure of the EPROM begins to occur when the chip is exposed to light with wavelengths shorter than approximately 4,000 angstroms. Since sunlight and fluorescent lighting have wavelengths in this range, exposure to these light sources over an extended time (about 1 week in sunlight, or 3 years in room level fluorescent lighting) could cause inadvertent erasure. **For this and secondary effects, it is recommended that an opaque label be placed over the window.** For elevated temperature or environments where solvents are being used, apply Kapton tape Fluorglas part number 2345–5, or equivalent.

The recommended erasure procedure is exposure to ultraviolet light (at 2537 angstroms) to an integrated dose of at least 15W-s/cm^2. Exposing the EPROM to an ultraviolet lamp of 12,000μW/cm^2 rating for 90 to 120 minutes, at a distance of about 1 inch, should be sufficient.

Erasure leaves the array in an all 1s state.

™Trademark phrase of Intel Corporation.

Security Bits

With none of the security bits programmed the code in the program memory can be verified. When only security bit 1 (see Table 5) is programmed, MOVC instructions executed from external program memory are disabled from fetching code bytes from the internal memory. All further programming of the EPROM is disabled. When security bits 1 and 2 are programmed, in addition to the above, verify mode is disabled. When all three security bits are programmed, all of the conditions above apply and all external program memory execution is disabled. (See Table 6.)

Table 5. EPROM Programming Modes

MODE	$\overline{\text{RST}}$	$\overline{\text{PSEN}}$	ALE/$\overline{\text{PROG}}$	$\overline{\text{EA}}$/V_{PP}	P1.0	P1.1	P1.2	P1.3	P1.4
Read signature	0	0	1	1	0	0	0	0	0
Program code data	0	0	0*	V_{PP}	0	1	1	1	1
Verify code data	0	0	1	1	0	0	1	1	0
Pgm security bit 1	0	0	0*	V_{PP}	1	1	1	1	1
Pgm security bit 2	0	0	0*	V_{PP}	1	1	0	0	1
Pgm security bit 3	0	0	0*	V_{PP}	1	0	1	0	1
Verify security bits	0	0	1	1	0	0	0	1	0

NOTES:
1. '0' = Valid low for that pin, '1' = valid high for that pin.
2. V_{PP} = 12.75V ±0.25V.
3. V_{CC} = 5V±10% during programming and verification.

* ALE/$\overline{\text{PROG}}$ receives 5 programming pulses (only for user array; 25 pulses for encryption or security bits) while V_{PP} is held at 12.75V. Each programming pulse is low for 100µs (±10µs) and high for a minimum of 10µs.

Table 6. Program Security Bits

PROGRAM LOCK BITS				PROTECTION DESCRIPTION
	SB1	SB2	SB3	
1	U	U	U	No Program Security features enabled.
2	P	U	U	MOVC instructions executed from external program memory are disabled from fetching code bytes from internal memory and further programming of the EPROM is disabled.
3	P	P	U	Same as 2, also verify is disabled.
4	P	P	P	Same as 3, external execution is disabled. Internal data RAM is not extermally accessible.

NOTES:
1. P – programmed. U – unprogrammed.
2. Any other combination of the security bits is not defined.

ROM CODE SUBMISSION

When submitting ROM code for the XA-G1, the following must be specified:

1. 8k byte user ROM data
2. ROM security bits.
3. Watchdog configuration

ADDRESS	CONTENT	BIT(S)	COMMENT
0000H to 1FFFH	DATA	7:0	User ROM Data
8020H	SEC	0	ROM Security Bit 1
8020H	SEC	1	ROM Security Bit 2 0 = enable security 1 = disable security
8020H	SEC	3	ROM Security Bit 3 0 = enable security 1 = disable security

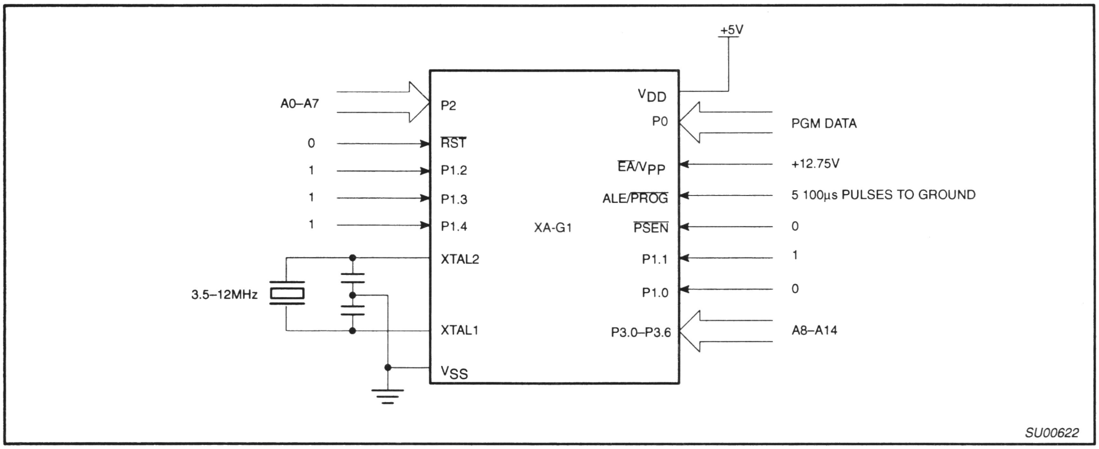

+5V
A0–A7
P2
VDD
P0
PGM DATA
0
RST
1
P1.2
EA/VPP
+12.75V
1
P1.3
ALE/PROG
5 100µs PULSES TO GROUND
1
P1.4
XA-G1
PSEN
0
XTAL2
P1.1
1
3.5–12MHz
P1.0
0
XTAL1
P3.0–P3.6
A8–A14
VSS
SU00622

Figure 24. Programming Configuration for XA-G1

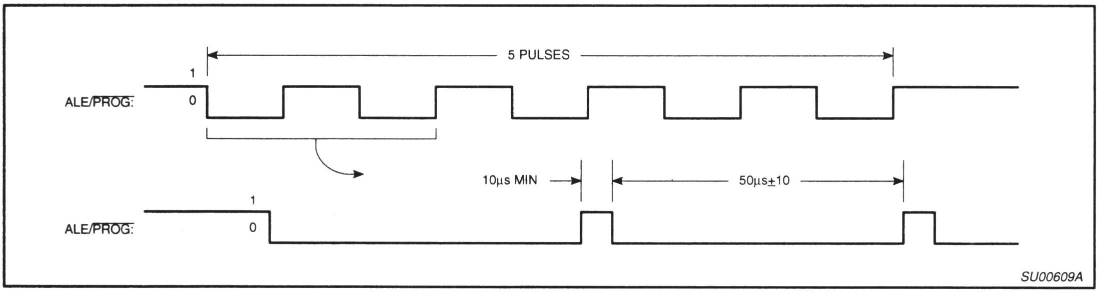

5 PULSES
1
ALE/PROG
0
10µs MIN
50µs±10
1
ALE/PROG
0
SU00609A

Figure 25. PROG Waveform

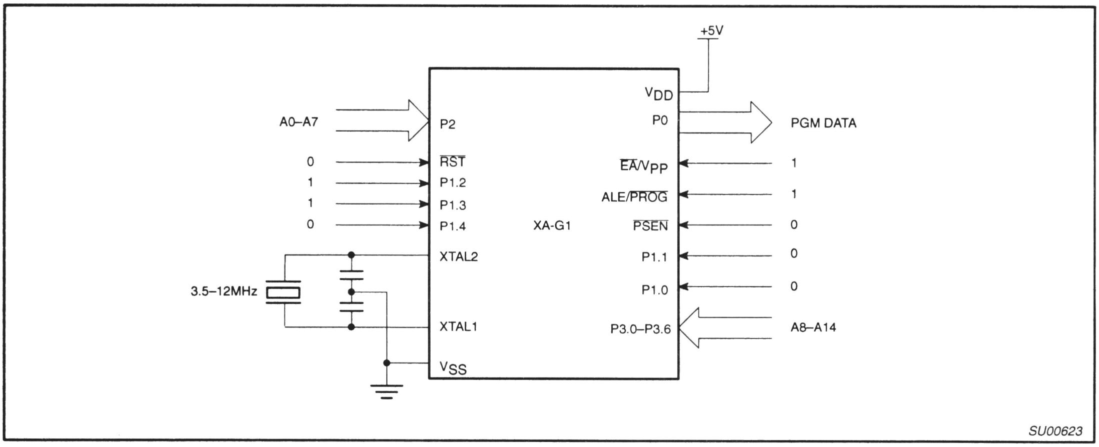

+5V
A0–A7
P2
VDD
P0
PGM DATA
0
RST
1
P1.2
EA/VPP
1
1
P1.3
ALE/PROG
1
0
P1.4
XA-G1
PSEN
0
XTAL2
P1.1
0
3.5–12MHz
P1.0
0
XTAL1
P3.0–P3.6
A8–A14
VSS
SU00623

Figure 26. Program Verification for XA-G1

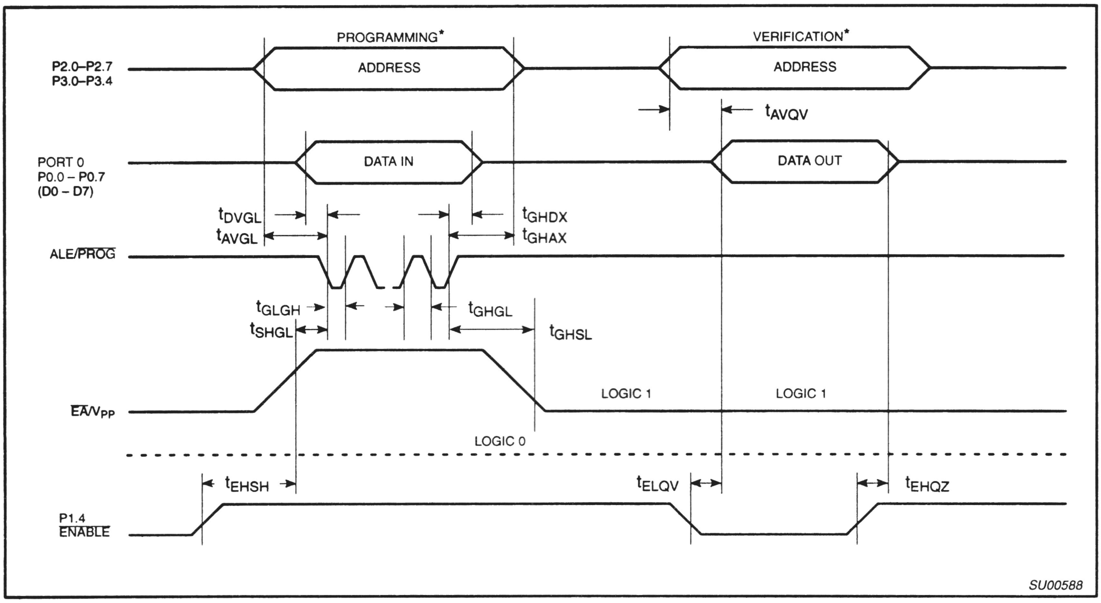
EPROM PROGRAMMING AND VERIFICATION CHARACTERISTICS

T_{amb} = 21°C to +27°C, V_{DD} = 5V±10%, V_{SS} = 0V (See Figure 27)

SYMBOL	PARAMETER	MIN	MAX	UNIT
V_{PP}	Programming supply voltage	12.5	13.0	V
I_{PP}	Programming supply current		50 [1]	mA
$1/t_{CL}$	Oscillator frequency	3.5	12	MHz
t_{AVGL}	Address setup to $\overline{PROG}$ low	$48t_{CL}$		
t_{GHAX}	Address hold after $\overline{PROG}$	$48t_{CL}$		
t_{DVGL}	Data setup to $\overline{PROG}$ low	$48t_{CL}$		
t_{GHDX}	Data hold after $\overline{PROG}$	$48t_{CL}$		
t_{EHSH}	P2.7 ($\overline{ENABLE}$) high to V_{PP}	$48t_{CL}$		
t_{SHGL}	V_{PP} setup to $\overline{PROG}$ low	10		µs
t_{GHSL}	V_{PP} hold after $\overline{PROG}$	10		µs
t_{GLGH}	$\overline{PROG}$ width	40	60	µs
t_{AVQV}	Address to data valid		$48t_{CL}$	
t_{ELQV}	$\overline{ENABLE}$ low to data valid		$48t_{CL}$	
t_{EHQZ}	Data float after $\overline{ENABLE}$	0	$48t_{CL}$	
t_{GHGL}	$\overline{PROG}$ high to $\overline{PROG}$ low	10		µs

NOTE:

1. Not tested.


PROGRAMMING*
VERIFICATION*
P2.0–P2.7
P3.0–P3.4
ADDRESS
ADDRESS
t_{AVQV}
PORT 0
P0.0 – P0.7
(D0 – D7)
DATA IN
DATA OUT
t_{DVGL}
t_{AVGL}
t_{GHDX}
t_{GHAX}
ALE/$\overline{PROG}$
t_{GLGH}
t_{SHGL}
t_{GHGL}
t_{GHSL}
$\overline{EA}/V_{PP}$
LOGIC 1
LOGIC 1
LOGIC 0
t_{EHSH}
t_{ELQV}
t_{EHQZ}
P1.4
$\overline{ENABLE}$
SU00588

NOTE:

* FOR PROGRAMMING CONDITIONS SEE FIGURE 25.
FOR VERIFICATION CONDITIONS SEE FIGURE 26.

Figure 27. EPROM Programming and Verification

CMOS single-chip 16-bit microcontroller

XA-G2

FAMILY DESCRIPTION

The Philips Semiconductors XA (eXtended Architecture) family of 16-bit single-chip microcontrollers is powerful enough to easily handle the requirements of high performance embedded applications, yet inexpensive enough to compete in the market for high-volume, low-cost applications.

The XA family provides an upward compatibility path for 80C51 users who need higher performance and 64k or more of program memory. Existing 80C51 code can also be easily be translated to run on XA microcontrollers.

The performance of the XA architecture supports the comprehensive bit-oriented operations of the 80C51 while incorporating support for multi-tasking operating systems and high-level languages such as C. The speed of the XA architecture, at 10 to 100 times that of the 80C51, gives designers and easy path to truly high performance embedded control.

The XA architecture supports:

- Upward compatibility with the 80C51 architecture
- 16-bit fully static CPU with a 24-bit program and data address range
- Eight 16-bit CPU registers each capable of performing all arithmetic and logic operations as well as acting as memory pointers. Operations may also be performed directly to memory.
- Both 8-bit and 16-bit CPU registers, each capable of performing all arithmetic and logic operations.
- An enhanced instruction set that includes bit intensive logic operations and fast signed or unsigned 16 × 16 multiply and 32 / 16 divide
- Instruction set tailored for high level language support
- Multi-tasking and real-time executives that include up to 32 vectored interrupts, up to 16 software traps, segmented data memory, and banked registers to support context switching
- Low power operation, which is intrinsic to the XA architecture includes power-down and idle modes.

More detailed information on the core is available in the XA User Guide.

SPECIFIC FEATURES OF THE XA-G2

- 20-bit address range, 1 megabyte each program and data space. (Note that the XA architecture supports up to 24 bit addresses.)
- 2.7V to 5.5V operation
- 16K bytes on-chip EPROM/ROM program memory
- 512 bytes of on-chip data RAM
- Three counter/timers with enhanced features (equivalent to 80C51 T0, T1, and T2)
- Watchdog timer
- Two enhanced UARTs
- Four 8-bit I/O ports with 4 programmable output configurations
- 44-pin PLCC and 44-pin LQFP packages

ORDERING INFORMATION

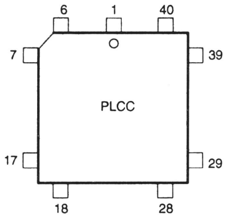
ROM	EPROM[1]		TEMPERATURE RANGE °C AND PACKAGE	FREQ (MHz)	DRAWING NUMBER
P51XAG23KB BD	P51XAG27KB BD	OTP	0 to +70, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG23KB A	P51XAG27KB A	OTP	0 to +70, Plastic Leaded Chip Carrier	30	SOT187–2
	P51XAG27KB KA	UV	0 to +70, Ceramic Leaded Chip Carrier	30	1472A
P51XAG23KF BD	P51XAG27KF BD	OTP	–40 to +85, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG23KF A	P51XAG27KF A	OTP	–40 to +85, Plastic Leaded Chip Carrier	30	SOT187–2
	P51XAG27KF KA	UV	–40 to +85, Ceramic Leaded Chip Carrier	30	1472A

NOTE:

1. OTP = One Time Programmable EPROM. UV = Erasable EPROM.

PIN CONFIGURATIONS

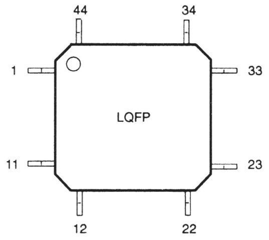
44-Pin PLCC Package



Pin	Function	Pin	Function
1	V_{SS}	23	V_{DD}
2	P1.0/A0/$\overline{\text{WRH}}$	24	P2.0/A12D8
3	P1.1/A1	25	P2.1/A13D9
4	P1.2/A2	26	P2.2/A14D10
5	P1.3/A3	27	P2.3/A15D11
6	P1.4/RxD1	28	P2.4/A16D12
7	P1.5/TxD1	29	P2.5/A17D13
8	P1.6/T2	30	P2.6/A18D14
9	P1.7/T2EX	31	P2.7/A19D15
10	$\overline{\text{RST}}$	32	$\overline{\text{PSEN}}$
11	P3.0/RxD0	33	ALE/$\overline{\text{PROG}}$
12	NC	34	NC
13	P3.1/TxD0	35	$\overline{\text{EA}}$/V_{PP}/WAIT
14	P3.2/$\overline{\text{INT0}}$	36	P0.7/A11D7
15	P3.3/$\overline{\text{INT1}}$	37	P0.6/A10D6
16	P3.4/T0	38	P0.5/A9D5
17	P3.5/T1/BUSW	39	P0.4/A8D4
18	P3.6/$\overline{\text{WRL}}$	40	P0.3/A7D3
19	P3.7/$\overline{\text{RD}}$	41	P0.2/A6D2
20	XTAL2	42	P0.1/A5D1
21	XTAL1	43	P0.0/A4D0
22	V_{SS}	44	V_{DD}

SU00525

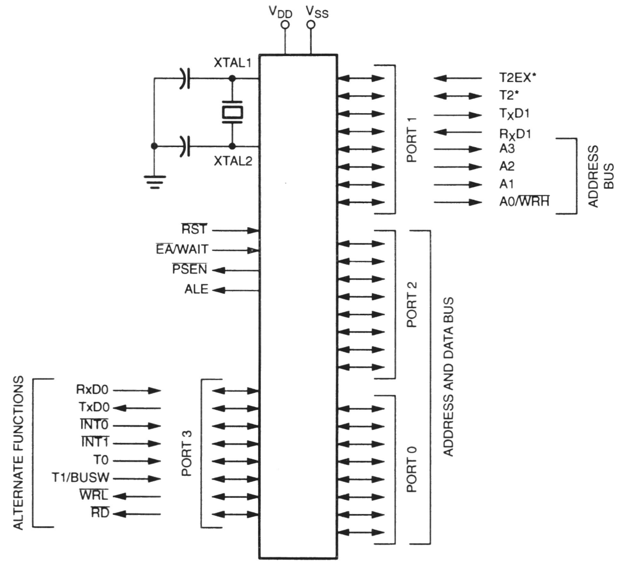
44-Pin LQFP Package



Pin	Function	Pin	Function
1	P1.5/TxD1	23	P2.5/A17D13
2	P1.6/T2	24	P2.6/A18D14
3	P1.7/T2EX	25	P2.7/A19D15
4	$\overline{\text{RST}}$	26	$\overline{\text{PSEN}}$
5	P3.0/RxD0	27	ALE/$\overline{\text{PROG}}$
6	NC	28	NC
7	P3.1/TxD0	29	$\overline{\text{EA}}$/V_{PP}/WAIT
8	P3.2/$\overline{\text{INT0}}$	30	P0.7/A11D7
9	P3.3/$\overline{\text{INT1}}$	31	P0.6/A10D6
10	P3.4/T0	32	P0.5/A9D5
11	P3.5/T1/BUSW	33	P0.4/A8D4
12	P3.6/$\overline{\text{WRL}}$	34	P0.3/A7D3
13	P3.7/$\overline{\text{RD}}$	35	P0.2/A6D2
14	XTAL2	36	P0.1/A5D1
15	XTAL1	37	P0.0/A4D0
16	V_{SS}	38	V_{DD}
17	V_{DD}	39	V_{SS}
18	P2.0/A12D8	40	P1.0/A0/$\overline{\text{WRH}}$
19	P2.1/A13D9	41	P1.1/A1
20	P2.2/A14D10	42	P1.2/A2
21	P2.3/A15D11	43	P1.3/A3
22	P2.4/A16/D12	44	P1.4/RxD1

SU00580

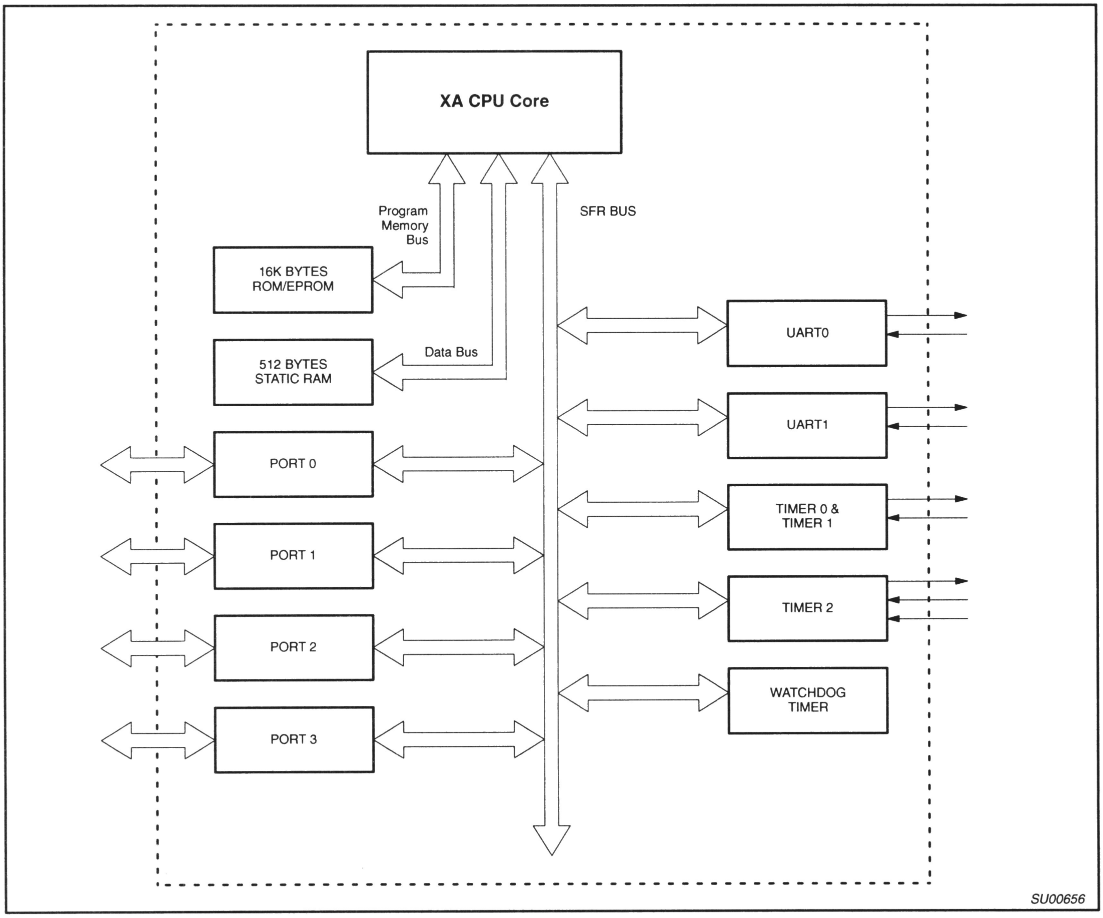
LOGIC SYMBOL


V_{DD}
V_{SS}
XTAL1
XTAL2
RST
EA/WAIT
PSEN
ALE
PORT 1
T2EX*
T2*
T_XD1
R_XD1
A3
A2
A1
A0/WRH
ADDRESS BUS
PORT 2
ADDRESS AND DATA BUS
PORT 0
PORT 3
ALTERNATE FUNCTIONS
RxD0
TxD0
INT0
INT1
T0
T1/BUSW
WRL
RD

* NOT AVAILABLE ON 40-PIN DIP PACKAGE

SU00526

BLOCK DIAGRAM


XA CPU Core
Program Memory Bus
SFR BUS
16K BYTES ROM/EPROM
512 BYTES STATIC RAM
Data Bus
PORT 0
PORT 1
PORT 2
PORT 3
UART0
UART1
TIMER 0 & TIMER 1
TIMER 2
WATCHDOG TIMER
SU00656

PIN DESCRIPTIONS

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
V_{SS}	1, 22	16	I	**Ground:** 0V reference.
V_{DD}	23, 44	17	I	**Power Supply:** This is the power supply voltage for normal, idle, and power down operation.
P0.0 – P0.7	43–36	37–30	I/O	**Port 0:** Port 0 is an 8-bit I/O port with a user-configurable output type. Port 0 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 0 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used, Port 0 becomes the multiplexed low data/instruction byte and address lines 4 through 11. Port 0 also outputs the code bytes during program verification and receives code bytes during EPROM programming.
P1.0 – P1.7	2–9	40–44, 1–3	I/O	**Port 1:** Port 1 is an 8-bit I/O port with a user-configurable output type. Port 1 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 1 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 1 also provides special functions as described below.
	2	40	O	**A0/$\overline{\text{WRH}}$:** Address bit 0 of the external address bus when the external data bus is configured for an 8 bit width. When the external data bus is configured for a 16 bit width, this pin becomes the high byte write strobe.
	3	41	O	**A1:** Address bit 1 of the external address bus.
	4	42	O	**A2:** Address bit 2 of the external address bus.
	5	43	O	**A3:** Address bit 3 of the external address bus. Port 1 also provides various special functions as described below.
	6	44	I	**RxD1 (P1.4):** Receiver input for serial port 1.
	7	1	O	**TxD1 (P1.5):** Transmitter output for serial port 1.
	8	2	I	**T2 (P1.6):** Timer/counter 2 external count input/clockout.
	9	3	I	**T2EX (P1.7):** Timer/counter 2 reload/capture/direction control
P2.0 – P2.7	24–31	18–25	I/O	**Port 2:** Port 2 is an 8-bit I/O port with a user-configurable output type. Port 2 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 2 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used in 16-bit mode, Port 2 becomes the multiplexed high data/instruction byte and address lines 12 through 19. When the external program/data bus is used in 8-bit mode, the number of address lines that appear on port 2 is user programmable. Port 2 also receives the low-order address byte during program memory verification.
P3.0 – P3.7	11, 13–19	5, 7–13	I/O	**Port 3:** Port 3 is an 8-bit I/O port with a user configurable output type. Port 3 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. the operation of port 3 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 3 pins receive the high order address bits during EPROM programming and verification. Port 3 also provides various special functions as described below.
	11	5	I	**RxD0 (P3.0):** Receiver input for serial port 0.
	13	7	O	**TxD0 (P3.1):** Transmitter output for serial port 0.
	14	8	I	**$\overline{\text{INT0}}$ (P3.2):** External interrupt 0 input.
	15	9	I	**$\overline{\text{INT1}}$ (P3.3):** External interrupt 1 input.
	16	10	I/O	**T0 (P3.4):** Timer 0 external input, or timer 0 overflow output.
	17	11	I/O	**T1/BUSW (P3.5):** Timer 1 external input, or timer 1 overflow output. The value on this pin is latched as the external reset input is released and defines the default external data bus width (BUSW).
	18	12	O	**$\overline{\text{WRL}}$ (P3.6):** External data memory low byte write strobe.
	19	13	O	**$\overline{\text{RD}}$ (P3.7):** External data memory read strobe.

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
$\overline{\text{RST}}$	10	4	I	**Reset:** A low on this pin resets the microcontroller, causing I/O ports and peripherals to take on their default states, and the processor to begin execution at the address contained in the reset vector. Refer to the section on Reset for details.
ALE/$\overline{\text{PROG}}$	33	27	I/O	**Address Latch Enable/Program Pulse:** A high output on the ALE pin signals external circuitry to latch the address portion of the multiplexed address/data bus. A pulse on ALE occurs only when it is needed in order to process a bus cycle. During EPROM programming, this pin is used as the program pulse input.
$\overline{\text{PSEN}}$	32	26	O	**Program Store Enable:** The read strobe for external program memory. When the microcontroller accesses external program memory, $\overline{\text{PSEN}}$ is driven low in order to enable memory devices. $\overline{\text{PSEN}}$ is only active when external code accesses are performed.
$\overline{\text{EA}}$/WAIT/ V_{PP}	35	29	I	**External Access/Wait/Programming Supply Voltage:** The $\overline{\text{EA}}$ input determines whether the internal program memory of the microcontroller is used for code execution. The value on the $\overline{\text{EA}}$ pin is latched as the external reset input is released and applies during later execution. When latched as a 0, external program memory is used exclusively, when latched as a 1, internal program memory will be used up to its limit, and external program memory used above that point. After reset is released, this pin takes on the function of bus Wait input. If Wait is asserted high during any external bus access, that cycle will be extended until Wait is released. During EPROM programming, this pin is also the programming supply voltage input.
XTAL1	21	15	I	**Crystal 1:** Input to the inverting amplifier used in the oscillator circuit and input to the internal clock generator circuits.
XTAL2	20	14	O	**Crystal 2:** Output from the oscillator amplifier.

SPECIAL FUNCTION REGISTERS

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
BCR	Bus configuration register	46A	—	—	—	WAITD	BUSD	BC2	BC1	BC0	Note 1
BTRH	Bus timing register low byte	469	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0	FF
BTRL	Bus timing register high byte	468	WM1	WM0	ALEW	—	CR1	CR0	CRA1	CRA0	EF
CS	Code segment	443									00
DS	Data segment	441									00
ES	extra segment	442									00
			33F	33E	33D	33C	33B	33A	339	338	
IEH*	Interrupt enable high byte	427	—	—	—	—	ETI1	ERI1	ETI0	ERI0	00
			337	336	335	334	333	332	331	330	
IEL*	Interrupt enable low byte	426	EA	—	—	ET2	ET1	EX1	ET0	EX0	00
IPA0	Interrupt priority 0	4A0	—	PT0			—	PX0			00
IPA1	Interrupt priority 1	4A1	—	PT1			—	PX1			00
IPA2	Interrupt priority 2	4A2	—	—			—	PT2			00
IPA4	Interrupt priority 4	4A4	—	PTI0			—	PRI0			00
IPA5	Interrupt priority 5	4A5	—	PTI1			—	PRI1			00
			387	386	385	384	383	382	381	380	
P0*	Port 0	430	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FF
			38F	38E	38D	38C	38B	38A	389	388	
P1*	Port 1	431	T2EX	T2	P1.5	P1.4	P1.3	P1.2	P1.1	WR1	FF
			397	396	395	394	393	392	391	390	
P2*	Port 2	432	P2.7	P2.6	P2.5	P2.4	TxD1	RxD1	P2.1	P2.0	FF

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES MSB							LSB	RESET VALUE
			39F	39E	39D	39C	39B	39A	399	398	
P3*	Port 3	433	RD	WR	T1	T0	INT1	INT0	TxD0	RxD0	FF
P0CFGA	Port 0 configuration A	470									Note 5
P1CFGA	Port 1 configuration A	471									Note 5
P2CFGA	Port 2 configuration A	472									Note 5
P3CFGA	Port 3 configuration A	473									Note 5
P0CFGB	Port 0 configuration B	4F0									Note 5
P1CFGB	Port 1 configuration B	4F1									Note 5
P2CFGB	Port 2 configuration B	4F2									Note 5
P3CFGB	Port 3 configuration B	4F3									Note 5
			227	226	225	224	223	222	221	220	
PCON*	Power control register	404	—	—	—	—	—	—	PD	IDL	00
			20F	20E	20D	20C	20B	20A	209	208	
PSWH*	Program status word (high byte)	401	SM	TM	RS1	RS0	IM3	IM2	IM1	IM0	Note 2
			207	206	205	204	203	202	201	200	
PSWL*	Program status word (low byte)	400	C	AC	—	—	—	V	N	Z	Note 2
			217	216	215	214	213	212	211	210	
PSW51*	80C51 compatible PSW	402	C	AC	F0	RS1	RS0	V	F1	P	Note 3
RTH0	Timer 0 extended reload, high byte	455									00
RTH1	Timer 1 extended reload, high byte	457									00
RTL0	Timer 0 extended reload, low byte	454									00
RTL1	Timer 1 extended reload, low byte	456									00
			307	306	305	304	303	302	301	300	
S0CON*	Serial port 0 control register	420	SM0_0	SM1_0	SM2_0	REN_0	TB8_0	RB8_0	TI_0	RI_0	00
			30F	30E	30D	30C	30B	30A	309	308	
S0STAT*	Serial port 0 extended status	421	—	—	—	—	FE0	BR0	OE0	STINT0	00
S0BUF	Serial port 0 buffer register	460									x
S0ADDR	Serial port 0 address register	461									00
S0ADEN	Serial port 0 address enable register	462									00
			327	326	325	324	323	322	321	320	
S1CON*	Serial port 1 control register	424	SM0_1	SM1_1	SM2_1	REN_1	TB8_1	RB8_1	TI_1	RI_1	00
			32F	32E	32D	32C	32B	32A	329	328	
S1STAT*	Serial port 1 extended status	425	—	—	—	—	FE1	BR1	OE1	STINT1	00
S1BUF	Serial port 1 buffer register	464									x
S1ADDR	Serial port 1 address register	465									00
S1ADEN	Serial port 1 address enable register	466									00
SCR	System configuration register	440	—	—	—	—	PT1	PT0	CM	PZ	00
			21F	21E	21D	21C	21B	21A	219	218	
SSEL*	Segment selection register	403	ESWEN	R6SEG	R5SEG	R4SEG	R3SEG	R2SEG	R1SEG	R0SEG	00
SWE	Software Interrupt Enable	47A	—	SWE7	SWE6	SWE5	SWE4	SWE3	SWE2	SWE1	00

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
			357	356	355	354	353	352	351	350	
SWR*	Software Interrupt Request	42A	—	SWR7	SWR6	SWR5	SWR4	SWR3	SWR2	SWR1	00
			2C7	2C6	2C5	2C4	2C3	2C2	2C1	2C0	
T2CON*	Timer 2 control register	418	TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2	00
			2CF	2CE	2CD	2CC	2CB	2CA	2C9	2C8	
T2MOD*	Timer 2 mode control	419	—	—	RCLK1	TCLK1	—	T2RD	T2OE	DCEN	00
TH2	Timer 2 high byte	459									00
TL2	Timer 2 low byte	458									00
T2CAPH	Timer 2 capture register, high byte	45B									00
T2CAPL	Timer 2 capture register, low byte	45A									00
			287	286	285	284	283	282	281	280	
TCON*	Timer 0 and 1 control register	410	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00
TH0	Timer 0 high byte	451									00
TH1	Timer 1 high byte	453									00
TL0	Timer 0 low byte	450									00
TL1	Timer 1 low byte	452									00
TMOD	Timer 0 and 1 mode register	45C	GATE	C/T	M1	M0	GATE	C/T	M1	M0	00
			28F	28E	28D	28C	28B	28A	289	288	
TSTAT*	Timer 0 and 1 extended status	411	—	—	—	—	T1RD	T1OE	T0RD	T0OE	00
			2FF	2FE	2FD	2FC	2FB	2FA	2F9	2F8	
WDCON*	Watchdog control register	41F	PRE2	PRE1	PRE0	—	—	WDRUN	WDTOF	—	Note 6
WDL	Watchdog timer reload	45F									00
WFEED1	Watchdog feed 1	45D									x
WFEED2	Watchdog feed 2	45E									x

NOTES:

* SFRs are bit addressable.

1. At reset, the BCR register is loaded with the binary value 0000 0a11, where “a” is the value on the BUSW pin.
2. SFR is loaded from the reset vector.
3. All bits except F1, F0, and P are loaded from the reset vector. Those bits are all 0.
4. Unimplemented bits in SFRs are X (unknown) at all times. Ones should not be written to these bits since they may be used for other purposes in future XA derivatives. The reset value shown for these bits is 0.
5. Port configurations default to quasi-bidirectional when the XA begins execution from internal code memory after reset, based on the condition found on the EA pin. Thus all PnCFGA registers will contain FF and PnCFGB registers will contain 00. When the XA begins execution using external code memory, the default configuration for pins that are associated with the external bus will be push-pull. The PnCFGA and PnCFGB register contents will reflect this difference.
6. The WDCON reset value is E6 for a Watchdog reset, E4 for all other reset causes.

XA-G2 TIMER/COUNTERS

The XA has two standard 16-bit enhanced Timer/Counters: Timer 0 and Timer 1. Additionally, it has a third 16-bit Up/Down timer/counter, T2. A central timing generator in the XA core provides the time-base for all XA Timers and Counters. The timer/event counters can perform the following functions:

- Measure time intervals and pulse duration
- Count external events
- Generate interrupt requests
- Generate PWM or timed output waveforms

All of the XA-G2 timer/counters (Timer 0, Timer 1 and Timer 2) can be independently programmed to operate either as timers or event counters via the C/T bit in the TnCON register. These timers may be dynamically read during program execution.

The base clock rate of all of the XA-G2 timers is user programmable. This applies to timers T0, T1, and T2 when running in timer mode (as opposed to counter mode), and the watchdog timer. The clock driving the timers is called TCLK and is determined by the setting of two bits (PT1, PT0) in the System Configuration Register (SCR). The frequency of TCLK may be selected to be the oscillator input divided by 4 (Osc/4), the oscillator input divided by 16 (Osc/16), or the oscillator input divided by 64 (Osc/64). This gives a range of possibilities for the XA timer functions, including baud rate generation, Timer 2 capture. Note that this single rate setting applies to all of the timers.

When timers T0, T1, or T2 are used in the counter mode, the register will increment whenever a falling edge (high to low transition) is detected on the external input pin corresponding to the timer clock. These inputs are sampled once every 2 oscillator cycles, so it can take as many as 4 oscillator cycles to detect a transition. Thus the maximum count rate that can be supported is Osc/4. The duty cycle of the timer clock inputs is not important, but any high or low state on the timer clock input pins must be present for 2 oscillator cycles before it is guaranteed to be “seen” by the timer logic.

Timer 0 and Timer 1

The “Timer” or “Counter” function is selected by control bits C/T in the special function register TMOD. These two Timer/Counters have four operating modes, which are selected by bit-pairs (M1, M0) in the TMOD register. Timer modes 1, 2, and 3 in XA are kept identical to the 80C51 timer modes for code compatibility. Only the mode 0 is replaced in the XA by a more powerful 16-bit auto-reload mode. This will give the XA timers a much larger range when used as time bases.

The recommended M1, M0 settings for the different modes are shown in Figure 2.

SCR Address:440
Not Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	—	—	PT1	PT0	CM	PZ

PT1	PT0	OPERATING
		Prescaler selection.
0	0	Osc/4
0	1	Osc/16
1	0	Osc/64
1	1	Reserved
CM		Compatibility Mode allows the XA to execute most translated 80C51 code on the XA. The XA register file must copy the 80C51 mapping to data memory and mimic the 80C51 indirect addressing scheme.
PZ		Page Zero mode forces all program and data addresses to 16-bits only. This saves stack space and speeds up execution but limits memory access to 64k.

SU00589

Figure 1. System Configuration Register (SCR)

TMOD Address:45C
Not Bit Addressable
Reset Value: 00H

MSB							LSB
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
TIMER 1				**TIMER 0**			

GATE	Gating control when set. Timer/Counter “n” is enabled only while “$\overline{\text{INTn}}$” pin is high and “TRn” control bit is set. When cleared Timer “n” is enabled whenever “TRn” control bit is set.
C/$\overline{T}$	Timer or Counter Selector cleared for Timer operation (input from internal system clock.) Set for Counter operation (input from “Tn” input pin).

M1	**M0**	**OPERATING**
0	0	16-bit auto-reload timer/counter
0	1	16-bit non-auto-reload timer/counter
1	0	8-bit auto-reload timer/counter
1	1	Dual 8-bit timer mode (timer 0 only)

SU00605

Figure 2. Timer/Counter Mode Control (TMOD) Register

New Enhanced Mode 0

For timers T0 or T1 the 13-bit count mode on the 80C51 (current Mode 0) has been replaced in the XA with a 16-bit auto-reload mode. Four additional 8-bit data registers (two per timer: RTHn and RTLn) are created to hold the auto-reload values. In this mode, the TH overflow will set the TF flag in the T2CON register and cause both the TL and TH counters to be loaded from the RTL and RTH registers respectively.

These new SFRs will also be used to hold the TL reload data in the 8-bit auto-reload mode (Mode 2) instead of TH.

Mode 1

Mode 1 is the 16-bit non-auto reload mode.

Mode 2

Mode 2 configures the Timer register as an 8-bit Counter (TLn) with automatic reload. Overflow from TLn not only sets TFn, but also reloads TLn with the contents of RTLn, which is preset by software. The reload leaves THn unchanged.

Mode 2 operation is the same for Timer/Counter 0.

Mode 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting TR1 = 0.

Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. TL0 uses the Timer 0 control bits: C/T, GATE, TR0, INT0, and TF0. TH0 is locked into a timer function and takes over the use of TR1 and TF1 from Timer 1. Thus, TH0 now controls the “Timer 1” interrupt.

Mode 3 is provided for applications requiring an extra 8-bit timer. When Timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt.

TCON Address:441
Bit Addressable
Reset Value: 00H

MSB							LSB
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

BIT	**SYMBOL**	**FUNCTION**
TCON.7	TF1	Timer 1 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.6	TR1	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TCON.5	TF0	Timer 0 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.4	TR0	Timer 0 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TCON.3	IE1	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.2	IT1	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
TCON.1	IE0	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.0	IT0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

SU00604

Figure 3. Timer/Counter Control (TCON) Register

T2CON Address:418
Bit Addressable
Reset Value: 00H

MSB LSB

TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2

BIT	SYMBOL	FUNCTION
T2CON.7	TF2	Timer 2 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or clearing the bit in software. TF2 will not be set when RCLK0, RCLK1, TCLK0, TCLK1 or T2OE=1.
T2CON.6	EXF2	Timer 2 external flag is set when a capture or reload occurs due to a negative transition on T2EX (and EXEN is set). This flag will cause a Timer 2 interrupt when this interrupt is enabled. EXF2 is cleared by software.
T2CON.5	RCLK0	Receive Clock Flag.
T2CON.4	TCLK0	Transmit Clock Flag. RCLK0 and TCLK0 are used to select Timer 2 overflow rate as a clock source for UART0.
T2CON.3	EXEN2	Timer 2 external enable flag allows a capture or reload to occure due to a negative transition on T2EX.
T2CON.2	TR2	Start=1/Stop=0 control for Timer 2.
T2CON.1	C/T2	Timer or counter select. 0=Internal timer 1=External event counter (falling edge triggered)
T2CON.0	IT0	Capture/Reload flag. If CP/RL2 & EXEN2=1 captures will occur on negative transitions of T2EX. If CP/RL2=0, EXEN2=1 auto reloads occur with either Timer 2 overflows or negative transitions at T2EX. If RCLK or TCLK=1 the timer is set to auto reload on Timer 2 overflow, this bit has no effect.

SU00606

Figure 4. Timer/Counter 2 Control (T2CON) Register

New Timer-Overflow Toggle Output

In the XA, the timer module now has two outputs, which toggle on overflow from the individual timers. The same device pins that are used for the T0 and T1 count inputs are also used for the new overflow outputs. An SFR bit (TnOE in the TSTAT register) is associated with each counter and indicates whether Port-SFR data or the overflow signal is output to the pin. These outputs could be used in applications for generating variable duty cycle PWM outputs (changing the auto-reload register values). Also variable frequency (Osc/8 to Osc/8,388,608) outputs could be achieved by adjusting the prescaler along with the auto-reload register values. With a 30.0MHz oscillator, this range would be 3.58Hz to 3.75MHz.

Timer T2

This is a 16-bit up or down counter, which can be operated as either a timer or event counter. It can be operated in one of three different modes (autoreload, capture or as the baud rate generator for either or both UARTs).

In the autoreload mode the Timer can be set to count up or down by setting or clearing the bit DCEN in the T2MOD Special Function Register. The SFR's T2CAPH and T2CAPL are used to reload the Timer upon overflow or to capture a 1-to-0 transition on the T2EX input (P1.7).

In the Capture mode Timer 2 can either set TF2 and generate an interrupt or capture its value. To capture Timer 2 in response to a 1-to-0 transition on the T2EX input, the EXEN2 bit in the T2CON must be set. Timer 2 is then captured in SFR's T2CAP2H and T2CAP2L.

As the baud rate generator, Timer 2 is selected by setting one of the RCLK and/or TCLK bits in T2CON or T2MOD. As the baud rate generator Timer 2 is incremented by TCLK.

Programmable Clock-Out

A 50% duty cycle clock can be programmed to come out on P1.6. This pin, besides being a regular I/O pin, has two alternate functions. It can be programmed (1) to input the external clock for Timer/Counter 2 or (2) to output a 50% duty cycle clock ranging from 3.58Hz to 3.75MHz at a 30MHz operating frequency.

To configure the Timer/Counter 2 as a clock generator, bit C/T2 (in T2CON) must be cleared and bit T20E in T2MOD must be set. Bit TR2 (T2CON.2) also must be set to start the timer.

The Clock-Out frequency depends on the oscillator frequency and the reload value of Timer 2 capture registers (TCAP2H, TCAP2L) as shown in this equation:

$$\frac{TCLK}{2 \times (65536 - TCAP2H, TCAP2L)}$$

In the Clock-Out mode Timer 2 roll-overs will not generate an interrupt. This is similar to when it is used as a baud-rate generator. It is possible to use Timer 2 as a baud-rate generator and a clock generator simultaneously. Note, however, that the baud-rate and the Clock-Out frequency will be the same.

TSTAT Address:411
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	—	—	T1RD	T1OE	T0RD	T0OE

BIT	SYMBOL	FUNCTION
TSTAT.3	T1RD	This bit is set, for Timer 1, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
TSTAT.2	T1OE	When 0, this bit allows the T1 pin to clock Timer 1 when in the counter mode. When 1, T1 acts as an output and toggles at every Timer 1 overflow.
TSTAT.1	T0RD	This bit is set, for Timer 0, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
TSTAT.0	T0OE	When 0, this bit allows the T0 pin to clock Timer 0 when in the counter mode. When 1, T0 acts as an output and toggles at every Timer 0 overflow.

SU00612A

Figure 5. Timer 0 And 1 Extended Status (TSTAT)

T2MOD Address:419
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	RCLK1	TCLK1	—	T2RD	T2OE	DCEN

BIT	SYMBOL	FUNCTION
T2MOD.5	RCLK1	Receive Clock Flag.
T2MOD.4	TCLK1	Transmit Clock Flag. RCLK1 and TCLK1 are used to select Timer 2 overflow rate as a clock source for UART1.
T2MOD.2	T2RD	This bit is set for Timer 2, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
T2MOD.1	T2OE	When 0, this bit allows the T2 pin to clock Timer 2 when in the counter mode. When 1, T2 acts as an output and toggles at every Timer 2 overflow.
T2MOD.0	DCEN	Controls count direction for Timer 2 in autoreload mode. DCEN=1 counter set to count up DCEN=0 counter set to count down

SU00610

Figure 6. Timer 2 Mode Control (T2MOD)

WATCHDOG TIMER

The watchdog timer subsystem protects the system from incorrect code execution by causing a system reset when the watchdog timer underflows as a result of a failure of software to feed the timer prior to the timer reaching its terminal count.

Watchdog Function

The watchdog consists of a programmable prescaler and the main timer. The prescaler derives its clock from the on-chip oscillator. The prescaler consists of a programmable TCLK followed by a 13 stage counter with taps from stage 6 through stage 13. This is shown in Figure 7. The tap selection is also programmable. The watchdog main counter is a down counter clocked (decremented) each time the programmable prescaler underflows. The watchdog generates an underflow signal (and is autoloaded) when the watchdog is at count 0 and the clock to decrement the watchdog occurs. The watchdog is 8 bits long and the autoload value can range from 0 to FFH. (The autoload value of 0 is permissible since the prescaler is cleared upon autoload).

This leads to the following user design equations. Definitions :t_{OSC} is the oscillator period, N is the selected prescaler tap value, W is the main counter autoload value, t_{MIN} is the minimum watchdog time-out value (when the autoload value is 0), t_{MAX} is the maximum time-out value (when the autoload value is FFH), t_D is the design time-out value.

$t_{MIN} = t_{OSC} \times 4 \times 64$ (W = 0)

$t_{MAX} = t_{OSC} \times 64 \times 8192 \times 256$ (W = 255)

$t_D = t_{MIN} \times 2^{PRESCALER} \times (W + 1)$
(where prescaler = 0, 1, 2, 3, 4, 5, 6, or 7)

The watchdog timer is not directly loadable by the user. Instead, the value to be loaded into the main timer is held in an autoload register or is part of the mask ROM programming. In order to cause the main timer to be loaded with the appropriate value, a special sequence of software action must take place. This operation is referred to as feeding the watchdog timer.

To feed the watchdog, two instructions must be sequentially executed successfully. No intervening SFR accesses are allowed, so interrupts should be disabled before feeding the watchdog. The instructions should move A5H to the WFEED1 register and then 5AH to the WFEED2 register. If WFEED1 is correctly loaded and WFEED2 is not correctly loaded, then an immediate watchdog reset will occur.

The software must be written so that a feed operation takes place every t_D seconds from the last feed operation. Some tradeoffs may need to be made. It is not advisable to include feed operations in minor loops or in subroutines unless the feed operation is a specific subroutine.

Watchdog Control Register (WDCON) (Bit Addressable)

The following bits of this register are read only in the ROM part when $\overline{EA}$ is high: PRE0, PRE1, and PRE2. That is, the register will reflect the mask programmed values. In the ROM part with $\overline{EA}$ high, these bits are taken from mask coded bits and are not readable by the program.

The reset values of the WDCON and WDL registers will be such that the watchdog timer has a timeout period of $4 \times 64 \times t_{OSC}$. The watchdog timer will not generate an interrupt. WDCON can be written by software only by executing a valid watchdog feed sequence.

The watchdog timer subsystem consists of a programmable 13-bit prescaler, and an 8-bit main timer. The main timer is clocked by a tap taken from one of the top 8-bits of the prescaler. The clock source for the prescaler is the same as TCLK (same as the clock source for the timers). Thus the main counter can be clocked as often as once every 64* TCLKs (see Table 1).

Table 1. Prescaler Select Values in WDCON

PRE2	PRE1	PRE0	DIVISOR
0	0	0	TCLK*32*2
0	0	1	TCLK*32*4
0	1	0	TCLK*32*8
0	1	1	TCLK*32*16
1	0	0	TCLK*32*32
1	0	1	TCLK*32*64
1	1	0	TCLK*32*128
1	1	1	TCLK*32*256

NOTE:
Where, $t_{CLK} = t_{OSC} \times 4, \times 16, \times 64$ (set in SCR).

Programming the Watchdog Timer

Both the EPROM and ROM devices have a set of SFRs for holding the watchdog autoload values and the control bits. The watchdog time-out flag is present in the watchdog control register and operates the same in all versions. In the EPROM device, the watchdog parameters (autoload value and control) are always taken from the SFRs. In the ROM device, the watchdog parameters can be mask programmed or taken from the SFRs. The selection to take the watchdog parameters from the SFRs or from the mask programmed values is controlled by $\overline{EA}$ (external access). When $\overline{EA}$ is high (internal ROM access), the watchdog parameters are taken from the mask programmed values. When $\overline{EA}$ is low (external access), the watchdog parameters are taken from the SFRs. The user should be able to leave code in his program which initializes the watchdog SFRs even though he has migrated to the mask ROM part. This allows no code changes from EPROM prototyping to ROM coded production parts.

Watchdog Detailed Operation

EPROM Device (and ROMless Operation: $\overline{EA}$ = 0)
In the ROMless operation (ROM part, $\overline{EA}$ = 0) and in the EPROM device, the watchdog operates in the following manner.

When external $\overline{RESET}$ is applied, the following takes place:

- Watchdog run control bit set to ON.
- Autoload register WDL set to 00 (min. count).
- Watchdog time-out flag cleared.
- Prescaler is cleared.
- Prescaler tap set to the lowest divide.
- Autoload takes place.

Note that when coming out of a hardware reset, the software should load the autoload registers and then feed the watchdog (cause an autoload). The watchdog will now be starting at a known point.

If the watchdog is running and happens to underflow at the time the external RESET is applied, the watchdog time-out flag will be cleared.

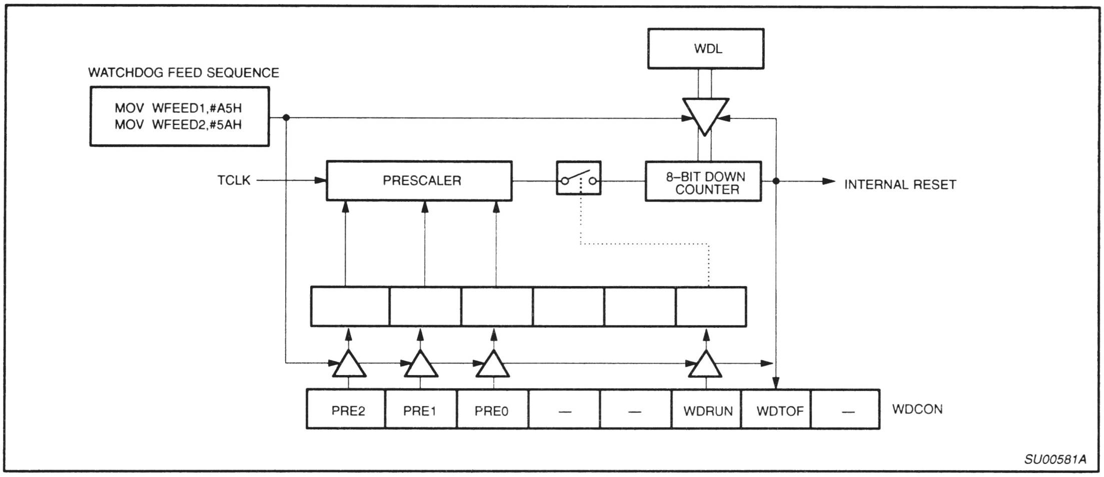


Figure 7. Watchdog Timer in XA-G2

When the watchdog underflows, the following action takes place (see Figure 7):

- Autoload takes place.
- Watchdog time-out flag is set
- Watchdog run bit unchanged.
- Autoload register unchanged.
- Prescaler tap unchanged.
- All other device action same as external reset.

Note that if the watchdog underflows, the program counter will be loaded from the reset vector as in the case of an internal reset. The watchdog time-out flag can be examined to determine if the watchdog has caused the reset condition. The watchdog time-out flag bit can be cleared by software.

WDCON Register Bit Definitions

WDCON.7	PRE2	Prescaler Select 2, reset to 1
WDCON.6	PRE1	Prescaler Select 1, reset to 1
WDCON.5	PRE0	Prescaler Select 0, reset to 1
WDCON.4	—	
WDCON.3	—	
WDCON.2	WDRUN, reset to 1	
WDCON.1	WDTOF	Timeout flag
WDCON.0	—	

UARTs

The XA-G2 includes 2 UART ports that are compatible with the enhanced UART used on the 8xC51FB. Baud rate selection is somewhat different due to the clocking scheme used for the XA timers.

Some other enhancements have been made to UART operation. The first is that there are separate interrupt vectors for each UART's transmit and receive functions. The second is double-buffering of the transmit register to allow time for interrupt processing without introducing inter-character gaps when tightly transmitted characters are required in the application. A break detect function has been added to the UART. This operates independently of the UART itself and provides a start-of-break status bit that the program may test. Finally, an Overrun Error flag has been added to detect missed characters in the received data stream.

Each UART rate is determined by either a fixed division of the oscillator (in UART modes 0 and 2) or by the timer 1 or timer 2 overflow rate (in UART modes 1 and 3).

The serial port receive and transmit registers are both accessed at Special Function Register SnBUF. Writing to SnBUF loads the transmit register, and reading SnBUF accesses a physically separate receive register.

The serial port can operate in 4 modes:

Mode 0: Serial I/O expansion mode. Serial data enters and exits through RxDn. TxDn outputs the shift clock. 8 bits are transmitted/received (LSB first). (The baud rate is fixed at 1/16 the oscillator frequency.)

Mode 1: Standard 8-bit UART mode. 10 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), and a stop bit (1). On receive, the stop bit goes into RB8 in Special Function Register SnCON. The baud rate is variable.

Mode 2: Fixed rate 9-bit UART mode. 11 bits are transmitted (through TxDn) or received (through RxDn): start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). On Transmit, the 9th data bit (TB8_8 in SnCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8_n. On receive, the 9th data bit goes into RB8_n in Special Function Register SnCON, while the stop bit is ignored. The baud rate is programmable to 1/32 of the oscillator frequency.

Mode 3: Standard 9-bit UART mode. 11 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). In fact, Mode 3 is the same as Mode 2 in all respects except baud rate. The baud rate in Mode 3 is variable.

In all four modes, transmission is initiated by any instruction that uses SnBUF as a destination register. Reception is initiated in Mode 0 by the condition RI_n = 0 and REN_n = 1. Reception is initiated in the other modes by the incoming start bit if REN_n = 1.

Serial Port Control Register
The serial port control and status register is the Special Function Register SnCON, shown in Figure 9. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8_n and RB8_n), and the serial port interrupt bits (TI_n and RI_n).

CLOCKING SCHEME/BAUD RATE GENERATION

The XA UARTS clock rates are determined by either a fixed division (modes 0 and 2) of the oscillator clock or by the Timer 1 or Timer 2 overflow rate (modes 1 and 3).

The clock for the UARTs in XA runs at 16x the Baud rate. If the timers are used as the source for Baud Clock, since maximum speed of timers/Baud Clock is Osc/4, the maximum baud rate is timer overflow divided by 16 i.e. Osc/64.

In Mode 0, it is fixed at Osc/16. In Mode 2, however, the fixed rate is Osc/32.

Pre-scaler for all Timers T0,1,2 controlled by PT1, PT0 bits in SCR	00	Osc/4
	01	Osc/16
	10	Osc/64
	11	reserved

Baud Rate for UART Mode 0:
Baud_Rate=Osc/16

Baud Rate calculation for UART Mode 1 and 3:
Baud_Rate=Timer_Rate/16*N

Timer_Rate=Osc/(N*(Timer_Range– Timer_Reload_Value))

where N=the TCLK prescaler value: 4, 16, or 64.
and Timer_Range= 256 for timer 1 in mode 2.
65536 for timer 1 in mode 0 and timer 2 in count up mode.

The timer reload value may be calculated as follows:
Timer_Reload_Value=Timer_Range–(Osc/(Baud_Rate*N*16))

NOTES:

1. The maximum baud rate for a UART in mode 1 or 3 is Osc/64.
2. The lowest possible baud rate (for a given oscillator frequency and N value) may be found by using a timer reload value of 0.
3. The timer reload value may never be larger than the timer range.
4. If a timer reload value calculation gives a negative or fractional result, the baud rate requested is not possible at the given oscillator frequency and N value.

Baud Rate for UART Mode 2:
Baud_Rate = Osc/32

Using Timer 2 to Generate Baud Rates

Timer T2 is a 16-bit up/down counter in XA. As a baud rate generator, timer 2 is selected as a clock source for either/both UART0 and UART1 transmitters and/or receivers by setting TCLKn and/or RCLKn in T2CON and T2MOD. As the baud rate generator, T2 is incremented as Osc/N where N=4, 16 or 64 depending on TCLK as programmed in the SCR bits PT1, and PTO. So, if T2 is the source of one UART, the other UART could be clocked by either T1 overflow or fixed clock, and the UARTs could run independently with different baud rates.

T2CON 0x418		bit5	bit4	
		RCLK0	TCLK0	

T2MOD 0x419		bit5	bit4	
		RCLK1	TCLK1	

When Timer 1 or 2 is the source for baud clock, the baud rate is given by
Baud Rate=Osc/16*1/Timer Overflow Rate

The timer T2 or T1 (16-bit mode) reload value is set by
Up-counter Mode
reload=65,536–[Osc/N)*1/Baud Rate]
where N=4,16 or 64

Down-counter Mode
reload=[(Osc/16N)*1/Baud Rate]
where N=4, 16, or 64

Prescaler Select for Timer Clock (TCLK)

SCR 0x440		bit3	bit2	
		PT1	PT0	

SnSTAT Address: S0STAT 421
S1STAT 425
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	—	—	FEn	BRn	OEn	STINTn

BIT	SYMBOL	FUNCTION
SnSTAT.3	FEn	Framing Error flag is set when the receiver fails to see a valid STOP bit at the end of the frame.
SnSTAT.2	BRn	Break Detect flag is set if a character is received with all bits (including STOP bit) being logic '0'. Thus it gives a "Start of Break Detect" on bit 8 for Mode 1 and bit 9 for Modes 2 and 3. The break detect feature operates independently of the UARTs and provides the START of Break Detect status bit that a user program may poll.
SnSTAT.1	OEn	Overrun Error flag is set if a new character is received in the receiver buffer while it is still full (before the software has read the previous character from the buffer), i.e., when bit 8 of a new byte is received while RI in SnCON is still set.
SnSTAT.0	STINTn	This flag must be set to enable any of the above status flags to generate a receive interrupt (RIn). The only way it can be cleared is by a software write to this register.

SU00607A

Figure 8. Serial Port Extended Status (SnSTAT) Register
(See also Figure 10 regarding Framing Error flag.)

INTERRUPT SCHEME

There are separate interrupt vectors for each UART's transmit and receive functions.

Table 2. Vector Locations for UARTs in XA

Vector Address	Interrupt Source	Arbitration
9CH – 9FH	Uart 1 Receiver	8
A0H – A3H	Uart 1 Transmitter	9
A4H – A7H	Uart 2 Receiver	10
A8H – ABH	Uart 2 Transmitter	11

NOTE:
The transmit and receive vectors could contain the same ISR address to work like a 8051 interrupt scheme

Error Handling, Status Flags and Break Detect

The UARTs in XA has the following error flags; see Figure 8.

Multiprocessor Communications

Modes 2 and 3 have a special provision for multiprocessor communications. In these modes, 9 data bits are received. The 9th one goes into RB8. Then comes a stop bit. The port can be programmed such that when the stop bit is received, the serial port interrupt will be activated only if RB8 = 1. This feature is enabled by setting bit SM2 in SCON. A way to use this feature in multiprocessor systems is as follows:

When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte differs from a data byte in that the 9th bit is 1 in an address byte and 0 in a data byte. With SM2 = 1, no slave will be interrupted by a data byte. An address byte, however, will interrupt all slaves, so that each slave can examine the received byte and see if it is being addressed. The addressed slave will clear its SM2 bit and prepare to receive the data bytes that will be coming. The slaves that weren't being addressed leave their SM2s set and go on about their business, ignoring the coming data bytes.

SM2 has no effect in Mode 0, and in Mode 1 can be used to check the validity of the stop bit although this is better done with the Framing Error (FE) flag. In a Mode 1 reception, if SM2 = 1, the receive interrupt will not be activated unless a valid stop bit is received.

Automatic Address Recognition

Automatic Address Recognition is a feature which allows the UART to recognize certain addresses in the serial bit stream by using hardware to make the comparisons. This feature saves a great deal of software overhead by eliminating the need for the software to examine every serial address which passes by the serial port. This feature is enabled by setting the SM2 bit in SCON. In the 9 bit UART modes, mode 2 and mode 3, the Receive Interrupt flag (RI) will be automatically set when the received byte contains either the "Given" address or the "Broadcast" address. The 9 bit mode requires that the 9th information bit is a 1 to indicate that the received information is an address and not data. Automatic address recognition is shown in Figure 11.

Using the Automatic Address Recognition feature allows a master to selectively communicate with one or more slaves by invoking the Given slave address or addresses. All of the slaves may be contacted by using the Broadcast address. Two special Function Registers are used to define the slave's address, SADDR, and the address mask, SADEN. SADEN is used to define which bits in the SADDR are to be used and which bits are "don't care". The SADEN mask can be logically ANDed with the SADDR to create the "Given" address which the master will use for addressing each of the slaves. Use of the Given address allows multiple slaves to be recognized while excluding others. The following examples will help to show the versatility of this scheme:

Slave 0 SADDR = 1100 0000
SADEN = 1111 1101
Given = 1100 00X0

Slave 1 SADDR = 1100 0000
SADEN = 1111 1110
Given = 1100 000X

In the above example SADDR is the same and the SADEN data is used to differentiate between the two slaves. Slave 0 requires a 0 in bit 0 and it ignores bit 1. Slave 1 requires a 0 in bit 1 and bit 0 is ignored. A unique address for Slave 0 would be 1100 0010 since slave 1 requires a 0 in bit 1. A unique address for slave 1 would be 1100 0001 since a 1 in bit 0 will exclude slave 0. Both slaves can be selected at the same time by an address which has bit 0 = 0 (for slave 0) and bit 1 = 0 (for slave 1). Thus, both could be addressed with 1100 0000.

In a more complex system the following could be used to select slaves 1 and 2 while excluding slave 0:

Slave 0 SADDR = 1100 0000
SADEN = 1111 1001
Given = 1100 0XX0

Slave 1 SADDR = 1110 0000
SADEN = 1111 1010
Given = 1110 0X0X

Slave 2 SADDR = 1110 0000
SADEN = 1111 1100
Given = 1110 00XX

In the above example the differentiation among the 3 slaves is in the lower 3 address bits. Slave 0 requires that bit 0 = 0 and it can be uniquely addressed by 1110 0110. Slave 1 requires that bit 1 = 0 and it can be uniquely addressed by 1110 and 0101. Slave 2 requires that bit 2 = 0 and its unique address is 1110 0011. To select Slaves 0 and 1 and exclude Slave 2 use address 1110 0100, since it is necessary to make bit 2 = 1 to exclude slave 2.

The Broadcast Address for each slave is created by taking the logical OR of SADDR and SADEN. Zeros in this result are teated as don't-cares. In most cases, interpreting the don't-cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR and SADEN are loaded with 0s. This produces a given address of all "don't cares" as well as a Broadcast address of all "don't cares". This effectively disables the Automatic Addressing mode and allows the microcontroller to use standard UART drivers which do not make use of this feature.

SnCON Address: S0CON 420
S1CON 424

Bit Addressable
Reset Value: 00H

MSB							LSB
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Where SM0, SM1 specify the serial port mode, as follows:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	shift register	fosc/ 16
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	fosc/32
1	1	3	9-bit UART	variable

BIT	SYMBOL	FUNCTION
SnCON.5	SM2	Enables the multiprocessor communication feature in Modes 2 and 3. In Mode 2 or 3, if SM2 is set to 1, then RI will not be activated if the received 9th data bit (RB8) is 0. In Mode 1, if SM2=1 then RI will not be activated if a valid stop bit was not received. In Mode 0, SM2 should be 0.
SnCON.4	REN	Enables serial reception. Set by software to enable reception. Clear by software to disable reception.
SnCON.3	TB8	The 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired.
SnCON.2	RB8	In Modes 2 and 3, is the 9th data bit that was received. In Mode 1, it SM2=0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
SnCON.1	TI	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
SnCON.0	RI	Receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

SU00597A

Figure 9. Serial Port Control (SnCON) Register

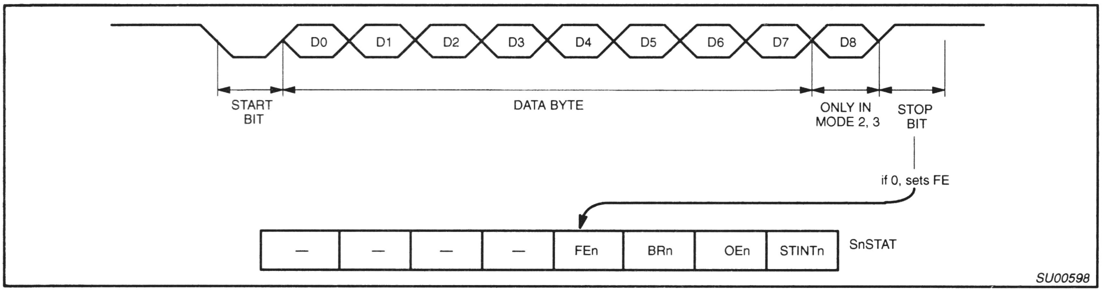
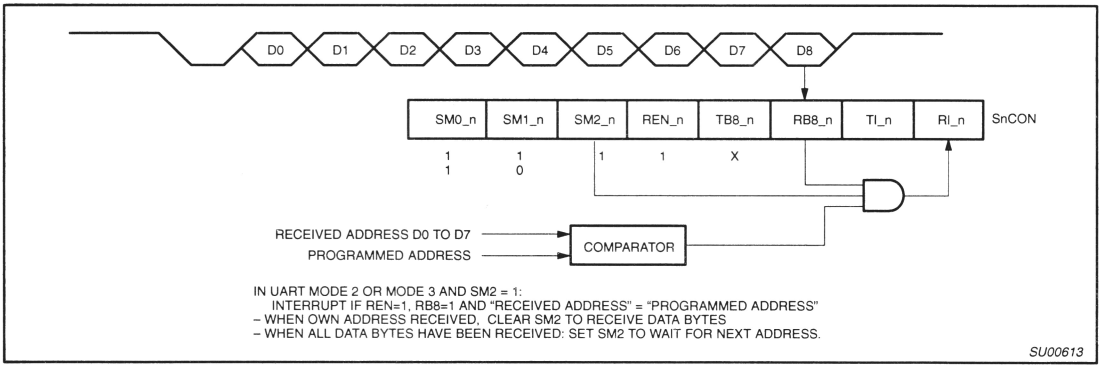

D0
D1
D2
D3
D4
D5
D6
D7
D8
START BIT
DATA BYTE
ONLY IN MODE 2, 3
STOP BIT
if 0, sets FE
—
—
—
—
FEn
BRn
OEn
STINTn
SnSTAT
SU00598

Figure 10. UART Framing Error Detection


D0
D1
D2
D3
D4
D5
D6
D7
D8
SM0_n
SM1_n
SM2_n
REN_n
TB8_n
RB8_n
TI_n
RI_n
SnCON
1
1
1
0
1
1
X
RECEIVED ADDRESS D0 TO D7
PROGRAMMED ADDRESS
COMPARATOR

IN UART MODE 2 OR MODE 3 AND SM2 = 1:
INTERRUPT IF REN=1, RB8=1 AND "RECEIVED ADDRESS" = "PROGRAMMED ADDRESS"
– WHEN OWN ADDRESS RECEIVED, CLEAR SM2 TO RECEIVE DATA BYTES
– WHEN ALL DATA BYTES HAVE BEEN RECEIVED: SET SM2 TO WAIT FOR NEXT ADDRESS.

SU00613

Figure 11. UART Multiprocessor Communication, Automatic Address Recognition

I/O PORT OUTPUT CONFIGURATION

Each I/O port pin on the XA-G2 can be user configured to one of 4 output types. The types are Quasi-bidirectional (essentially the same as standard 80C51 family I/O ports), Open-Drain, Push-Pull, and Off (high impedance). The default configuration after reset is Quasi-bidirectional. However, in the ROMless mode (the $\overline{EA}$ pin is low at reset), the port pins that comprise the external data bus will default to push-pull outputs.

I/O port output configurations are determined by the settings in port configuration SFRs. There are 2 SFRs for each port, called PnCFGA and PnCFGB, where "n" is the port number. One bit in each of the 2 SFRs relates to the output setting for the corresponding port pin, allowing any combination of the 2 output types to be mixed on those port pins. For instance, the output type of port 1 pin 3 is controlled by the setting of bit 3 in the SFRs P1CFGA and P1CFGB.

Table 3 shows the configuration register settings for the 4 port output types. The electrical characteristics of each output type may be found in the DC Characteristic table.

Table 3. Port Configuration Register Settings

PnCFGB	PnCFGA	Port Output Mode
0	0	Open Drain
0	1	Quasi-bidirectional
1	0	Off (high impedance)
1	1	Push-Pull

NOTE:
Mode changes may cause glitches to occur during transitions. When modifying both registers, WRITE instructions should be carried out consecutively.

EXTERNAL BUS

The external program/data bus on the XA-G2 allows for 8-bit or 16-bit bus width, and address sizes from 12 to 20 bits. The bus width is selected by an input at reset (see Reset Options below), while the address size is set by the program in a configuration register. If all off-chip code is selected (through the use of the $\overline{EA}$ pin), the initial code fetches will be done with the maximum address size (20 bits).

RESET

The device is reset whenever a logic "0" is applied to $\overline{RST}$ for at least 10 microseconds, placing a low level on the pin re-initializes the on-chip logic.

The duration of reset must be extended when power is initially applied or when using reset to exit power down mode. This is due to the need to allow the oscillator time to start up and stabilize. For most power supply ramp up conditions, this time is 10 milliseconds.

As it is brought high again, an exception is generated which causes the processor to jump to the address contained in the memory location 0000. The destination of the reset jump must be located in the first 64k of code address on power-up, all vectors are 16-bit values and so point to page zero addresses only. After a reset the RAM contents are indeterminate.

RESET OPTIONS

The $\overline{EA}$ pin is sampled on the rising edge of the $\overline{RST}$ pulse, and determines whether the device is to begin execution from internal or external code memory. $\overline{EA}$ pulled high configures the XA in single-chip mode. If $\overline{EA}$ is driven low, the device enters ROMless mode. After Reset is released, the $\overline{EA}$/WAIT pin becomes a bus wait signal for external bus transactions.

The BUSW/P3.5 pin is weakly pulled high while reset is asserted, allowing simple biasing of the pin with a resistor to ground to select the alternate bus width.

POWER REDUCTION MODES

The XA-G2 supports Idle and Power down modes of power reduction. The idle mode leaves some peripherals running to allow them to wake up the processor when an interrupt is generated. The power down mode stops the processor clock in order to absolutely minimize power. The processor can be made to exit power down mode via reset or one of the external interrupt inputs. In power down mode, the power supply voltage may be further reduced to the keep-alive voltage, retaining the RAM, register, and SFR values at the point where the power down mode was entered.

INTERRUPTS

The XA-G2 supports 31 maskable interrupts vectored interrupt sources. The maskable interrupts each have 16 priority levels and may be globally and/or individually enabled or disabled.

The XA defines four types of interrupts:

- **Exception Interrupts** – These are system level errors and other very important occurrences which include stack overflow, divide-by-0, and reset.
- **Event interrupts** – These are peripheral interrupts from devices such as UARTs, timers, and external interrupt inputs.
- **Software Interrupts** – These are equivalent of hardware interrupt, but are requested only under software control.
- **Trap Interrupts** – These are TRAP instructions, generally used to call system services in a multi-tasking system.

Exception interrupts, software interrupts, and trap interrupts are generally standard for XA derivatives and are detailed in the XA User Guide. Event interrupts tend to be different on different XA derivatives.

The XA-G2 supports a total of 9 maskable event interrupt sources (for the various XA-G2 peripherals), seven software interrupts, 5 exception interrupts (plus reset), and 16 traps. The maskable event interrupts share a global interrupt enable bit (the EA bit in the IEL register) and each also has a separate individual interrupt enable bit (in the IEL or IEH registers). Each event interrupt can be set to occur at one of 8 priority levels (levels 8 through 15) via bits in the Interrupt Priority (IP) registers, IPA0 through IPA5. Details of the priority scheme may be found in the XA User Guide.

The complete interrupt vector list for the XA-G2, including all 4 interrupt types, is shown in the following tables. The tables include the address of the vector for each interrupt, the related priority register bits (if any), and the arbitration ranking for that interrupt source. The arbitration ranking determines the order in which interrupts are processed if more than one interrupt of the same priority occurs simultaneously.

Table 4. Interrupt Vectors

EXCEPTION/TRAPS PRECEDENCE

DESCRIPTION	VECTOR ADDRESS	ARBITRATION RANKING
Reset (h/w, watchdog, s/w)	0000–0003	0 (High)
Breakpoint (h/w trap 1)	0004–0007	1
Trace (h/w trap 2)	0008–000B	1
Stack Overflow (h/w trap 3)	000C–000F	1
Divide by 0 (h/w trap 4)	0010–0013	1
User RETI (h/w trap 5)	0014–0017	1
TRAP 0– 15 (software)	0040–007F	1

EVENT INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY	ARBITRATION RANKING
External interrupt 0	IE0	0080–0083	EX0	IPA0.3–0	2
Timer 0 interrupt	TF0	0084–0087	ET0	IPA0.7–4	3
External interrupt 1	IE1	0088–008B	EX1	IPA1.3–0	4
Timer 1 interrupt	TF1	008C–008F	ET1	IPA1.7–4	5
Timer 2 interrupt	TF2	0090–0093	ET2	IPA2.3–0	6
Serial port 0 Rx	RI.0	00A0–00A3	ERI0	IPA4.3–0	7
Serial port 0 Tx	TI.0	00A4–00A7	ETI0	IPA4.7–4	8
Serial port 1 Rx	RI.1	00A8–00AB	ERI1	IPA5.3–0	9
Serial port 1 Tx	TI.1	00AC–00AF	ETI1	IPA5.7–4	10

SOFTWARE INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY
Software interrupt 1	SWR1	0100–0103	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010B	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010F	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0018–011B	SWE7	(fixed at 7)

ABSOLUTE MAXIMUM RATINGS

PARAMETER	RATING	UNIT
Operating temperature under bias	–55 to +125	°C
Storage temperature range	–65 to +150	°C
Voltage on $\overline{EA}$/V_{PP} pin to V_{SS}	0 to +13.0	V
Voltage on any other pin to V_{SS}	–0.5 to V_{DD} + 0.5V	V
Maximum I_{OL} per I/O pin	15	mA
Power dissipation (based on package heat transfer limitations, not device power consumption)	1.5	W

DC ELECTRICAL CHARACTERISTICS

V_{DD} = 5.0V ±10% to 3.0V ±10% unless otherwise specified;
T_{amb} = T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial, unless otherwise specified.

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS			UNIT
			MIN	TYP	MAX	
Supplies						
I_{DD}	Supply current operating	5.0V, 30MHz			100	mA
I_{ID}	Idle mode supply current	5.0V, 30MHz			25	mA
I_{PD}	Power-down current	5.0V, 3.0V		5	50	µA
V_{RAM}	RAM-keep-alive voltage	RAM-keep-alive voltage	1.5			V
V_{IL}	Input low voltage, except		–0.5		0.8	V
V_{IH}	Input high voltage, except XTAL1, RST	At 5.0V[1]	2.2			V
		At 3.0V[1]	2			V
V_{IH1}	Input high voltage to XTAL1, RST	For both 3.0 & 5.0V	$0.7V_{DD}$			V
V_{OL}	Output low voltage all ports, ALE, PSEN[5]	I_{OL} = 3.2mA, V_{DD} = 5.0V			0.8	V
		1.0mA, V_{DD} = 3.0V				V
V_{OH1}	Output high voltage all ports, ALE, PSEN[3]	I_{OH} = –100µA, V_{DD} = 5.0V	2.4			V
		I_{OH} = –30µA, V_{DD} = 3.0V	2.2			V
V_{OH2}	Output high voltage, ports P0–3, ALE, PSEN[4]	I_{OH} = 3.2mA, V_{DD} = 5.0V	2.4			V
		I_{OH} = 1mA, V_{DD} = 3.0V	2.2			V
C_{IO}	Input/Output pin capacitance[2]				15	pF
I_{IL}	Logical 0 input current, P0–3[8]	V_{IN} = 0.45V			–50	µA
I_{LI}	Input leakage current, P0–3[7]				±10	µA
I_{TL}	Logical 1 to 0 transition current all ports[6]	At 6V			–650	µA
		At 3V			–250	µA

NOTE:

1. Values are linear in between
2. Max. 15pF for ~EA/V_{PP}
3. Ports in Quasi bi-directional mode with weak pull-up
4. Ports in Push-Pull mode, both pull-up and pull-down assumed to be same strength
5. In all output modes
6. Port pins source a transition current when used in quasi-bidirectional mode and externally driven from 1 to 0. This current is highest when V_{IN} is approximately 2V.
7. Measured with port in high impedance output mode.
8. Measured with port in quasi-bidirectional output mode.
9. Load capacitance for port 0, ALE, and $\overline{PSEN}$ = 100pF, load capacitance for all other outputs = 80pF.
10. Under steady state (non-transient) conditions, I_{OL} must be extremely limited as follows:
    Maximum I_{OL} per port pin: 15mA (*NOTE: This is 85°C specification.)
    Maximum I_{OL} per 8-bit port: 26mA
    Maximum total I_{OL} for all output: 71mA
    If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.

AC ELECTRICAL CHARACTERISTICS

V_{DD} = 5.0V ±10% or 3.0V ±10%, T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial.

SYMBOL	PARAMETER	30MHz[1]			16MHz[2]			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
External Clock								
f_C	Oscillator frequency		30			16		MHz
$t_C = 1/f_C$	Clock period and CPU timing cycle		$1/f_C$			$1/f_C$		ns
t_{CHCX}	Clock high-time (60%–40% duty cycle)		$(t_C/2) * 0.4$			$(t_C/2) * 0.4$		ns
t_{CLCX}	Clock low-time (60%–40% duty cycle)		$(t_C/2) * 0.4$			$(t_C/2) * 0.4$		ns
t_{CLCH}	Clock rise-time		5			10		ns
t_{CHCL}	Clock fall-time		5			10		ns
Address Cycle								
t_{CRAR}	Delay from clock rising edge to ALE rising edge		24			38		ns
t_{LHLL}	ALE pulse width (programmable)		$(N+0.5) * t_C$			$(N+0.5) * t_C$		ns
t_{AVLL}	Address valid to ALE de-asserted (set-up)		t_{LLHL}			$t_{LLHL} - 4$		ns
t_{LLAX}	Address hold after ALE de-asserted		12			30		ns
Code Read Cycle								
t_{LLPL}	ALE de-asserted to PSEN active		$(t_C/2) + 10$			$(t_C/2) + 1$		ns
t_{AVIV}	Address valid to instruction valid (access time)		$(M*t_C) - 16$			$(M*t_C) - 21$		ns
t_{PLIV}	$\overline{\text{PSEN}}$ low to instruction valid		$(O*t_C) - 16$			$(O*t_C) - 21$		ns
t_{PLPH}	$\overline{\text{PSEN}}$ pulse width		$(O*t_C) - 5$			$O*t_C$		ns
t_{PXIX}	Instruction hold after $\overline{\text{PSEN}}$ de-asserted	0			0			ns
t_{PXIZ}	Bus 3-State after $\overline{\text{PSEN}}$ de-asserted		29			54		ns
t_{UAPH}	Hold time of unlatched port of address after $\overline{\text{PSEN}}$ is de-asserted.	0			0			ns
Data Read Cycle								
t_{RLRH}	$\overline{\text{RD}}$ pulse width		$(P*t_C) - 8$			$P*t_C$		ns
t_{LLRL}	ALE falling edge to $\overline{\text{RD}}$ falling edge		$(t_C/2) + 9$			$(t_C/2) + 1$		ns
t_{AVDV}	Data input valid after address valid (access time)		$(P*t_C) - 16$			$(P*t_C) - 21$		ns
t_{RLDV}	$\overline{\text{RD}}$ low to valid data in, enable time		$(P*t_C) - 16$			$(P*t_C) - 21$		ns
t_{RHDX}	Data hold time after $\overline{\text{RD}}$ de-asserted	0			0			ns
t_{RHDZ}	Bus 3-State after $\overline{\text{RD}}$ de-asserted		29			54		ns
t_{UARH}	Hold time of unlatched port of address after $\overline{\text{RD}}$ is de-asserted.	0			0			ns
Data Write Cycle								
t_{WLWH}	$\overline{\text{WR}}$ pulse width		$Q*t_C$			$Q*t_C$		ns
t_{LLWL}	ALE falling edge to $\overline{\text{WR}}$ asserted		$(t_C/2) + 5$			$(t_C/2) - 5$		ns
t_{QVWX}	Data valid before $\overline{\text{WR}}$ active (setup time)		$(R*t_C) - 7$			$(R*t_C) - 12$		ns
t_{WHQX}	Data hold time after $\overline{\text{WR}}$ rising edge		0			4		ns
t_{AVWL}	address valid to $\overline{\text{WR}}$ active		$(R*t_C) - 3$			$(R*t_C) - 8$		ns
t_{UAWH}	Hold time of unlatched part of address after $\overline{\text{WR}}$ is de-asserted		0			3		ns
WAIT Input								
t_{WTV}	Rising edge of Wait after $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{PSEN}}$ falling edge		$(S*t_C) - 15$			$(S*t_C) - 14$		ns
t_{WAIT}	CPU wait state period		t_C			t_C		ns

SYMBOL	PARAMETER	30MHz[1]			16MHz[2]			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
Input Pulse Width								
t_{PWI}	Pulse width for interrupt inputs		2^*t_C			2^*t_C		ns
t_{PWT}	Pulse width for timer inputs		2^*t_C			2^*t_C		ns
Shift Register								
t_{XLXL}	Serial port clock cycle time		16^*t_C			16^*t_C		ns
t_{QVXH}	Output data setup to clock rising edge		$(2^*t_C) - 30$			$(2^*t_C) - 40$		ns
t_{XHQX}	Output data hold to clock rising edge		$(2^*t_C) - 30$			$(2^*t_C) - 40$		ns
t_{XHDX}	Input data hold after clock rising edge		0			0		ns
t_{XHDV}	Clock rising edge to input data valid		30			40		ns

NOTES:
1. All values indicated for $V_{DD} = 5V \pm 10\%$. Typical values are for 20°C.
2. All values indicated for $V_{DD} = 3V \pm 10\%$. Typical values are for 20°C.
3. N = ALEW bit value
   M = burst mode code read clocks
   O = PSEN clocks
   P = RD clocks
   Q = WR clocks
   R = WR setup clocks

EXPLANATION OF THE AC SYMBOLS

Each timing symbol has five characters. The first character is always 't' (= time). The other characters, depending on their positions, indicate the name of a signal or the logical status of that signal. The designations are:

A – Address
C – Clock
D – Input data
H – Logic level high
I – Instruction (program memory contents)
L – Logic level low, or ALE
P – $\overline{\text{PSEN}}$
Q – Output data
R – $\overline{\text{RD}}$ signal
t – Time
U – Undefined
V – Valid
W – $\overline{\text{WR}}$ signal
X – No longer a valid logic level
Z – Float

Examples: t_{AVLL} = Time for address valid to ALE low.
t_{LLPL} =Time for ALE low to $\overline{\text{PSEN}}$ low.

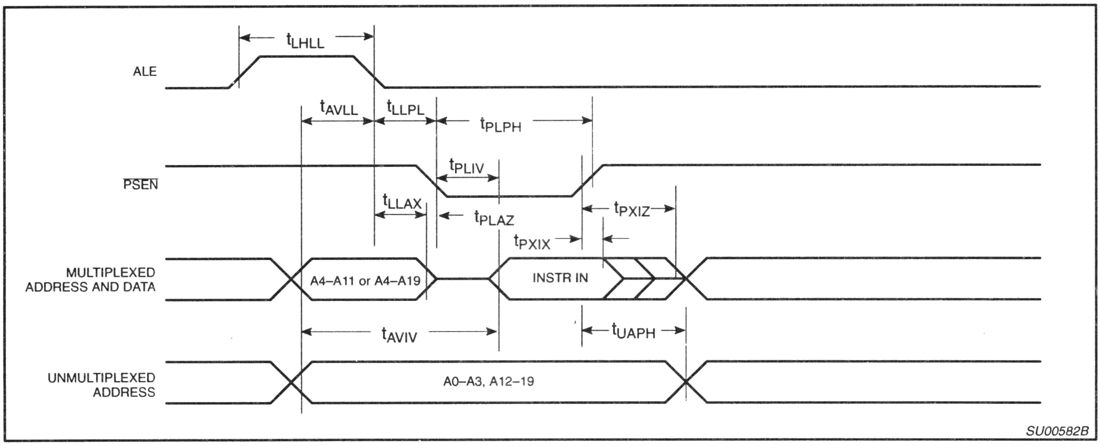

t_{LHLL}
ALE
t_{AVLL}
t_{LLPL}
t_{PLPH}
$\overline{\text{PSEN}}$
t_{PLIV}
t_{LLAX}
t_{PLAZ}
t_{PXIZ}
t_{PXIX}
MULTIPLEXED ADDRESS AND DATA
A4–A11 or A4–A19
INSTR IN
t_{AVIV}
t_{UAPH}
UNMULTIPLEXED ADDRESS
A0–A3, A12–19
SU00582B

Figure 12. External Program Memory Read Cycle

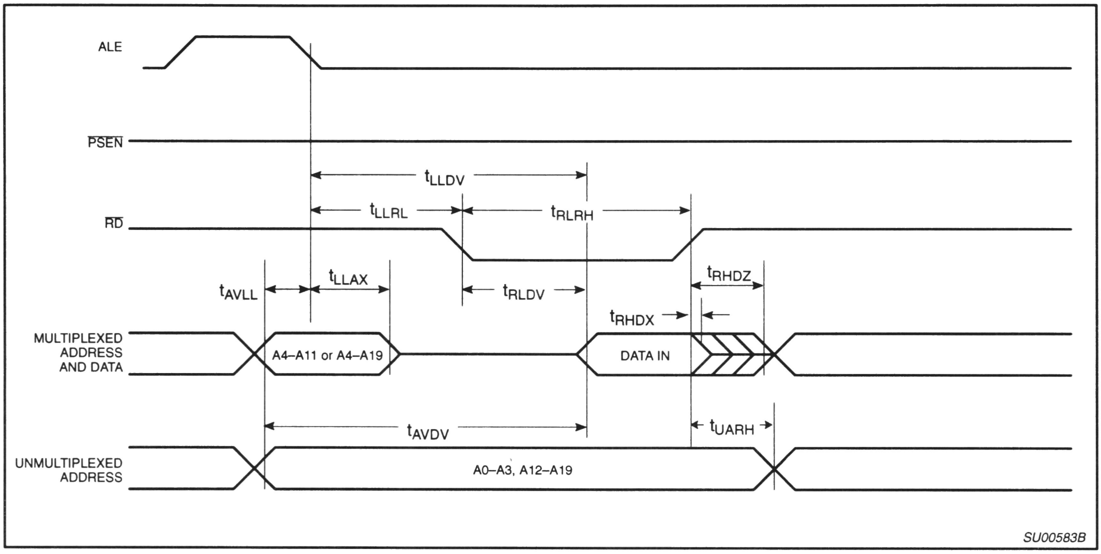

ALE
PSEN
RD
MULTIPLEXED ADDRESS AND DATA
UNMULTIPLEXED ADDRESS
t_{LLDV}
t_{LLRL}
t_{RLRH}
t_{AVLL}
t_{LLAX}
t_{RLDV}
t_{RHDZ}
t_{RHDX}
A4–A11 or A4–A19
DATA IN
t_{AVDV}
t_{UARH}
A0–A3, A12–A19
SU00583B

Figure 13. External Data Memory Read Cycle

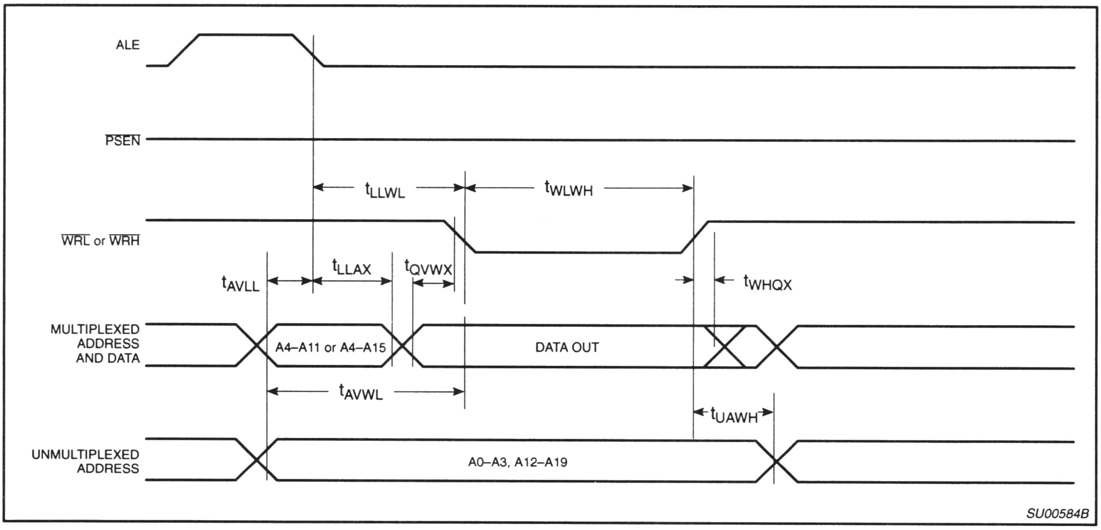

ALE
PSEN
WRL or WRH
MULTIPLEXED ADDRESS AND DATA
UNMULTIPLEXED ADDRESS
t_{LLWL}
t_{WLWH}
t_{AVLL}
t_{LLAX}
t_{QVWX}
t_{WHQX}
A4–A11 or A4–A15
DATA OUT
t_{AVWL}
t_{UAWH}
A0–A3, A12–A19
SU00584B

Figure 14. External Data Memory Write Cycle

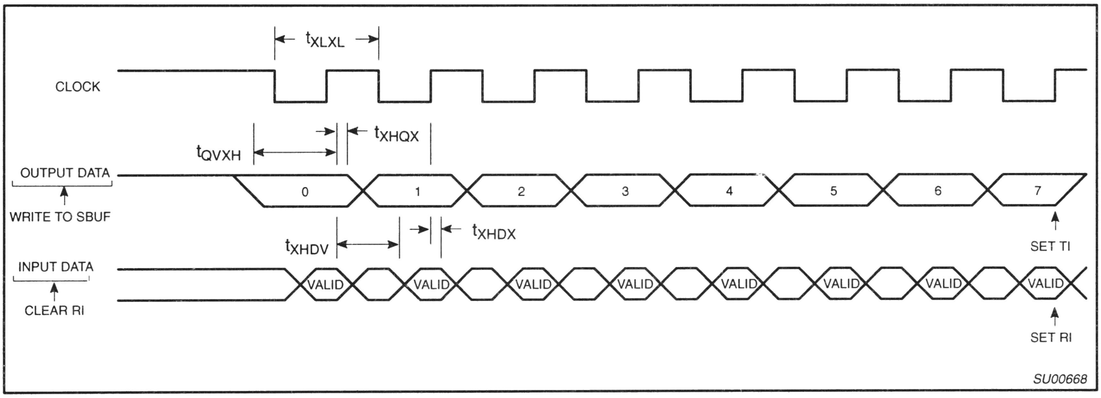

t_{XLXL}
CLOCK
t_{QVXH}
t_{XHQX}
OUTPUT DATA
WRITE TO SBUF
0
1
2
3
4
5
6
7
SET TI
t_{XHDV}
t_{XHDX}
INPUT DATA
CLEAR RI
VALID
SET RI
SU00668

Figure 15. Shift Register Mode Timing

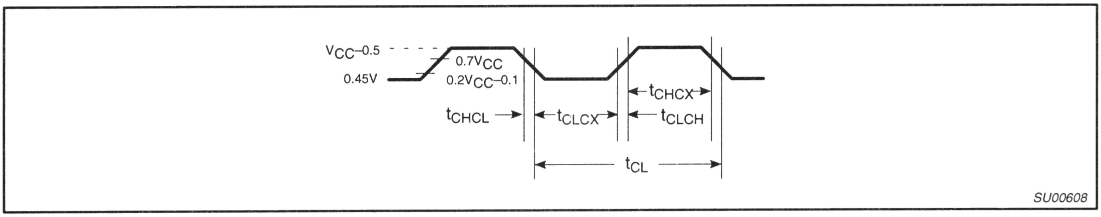
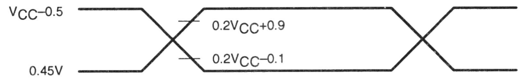

V_{CC}–0.5
0.45V
$0.7V_{CC}$
$0.2V_{CC}$–0.1
t_{CHCL}
t_{CLCX}
t_{CHCX}
t_{CLCH}
t_{CL}
SU00608

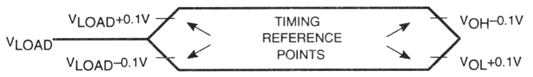
Figure 16. External Clock Drive


V_{CC}–0.5
0.45V
$0.2V_{CC}$+0.9
$0.2V_{CC}$–0.1

NOTE:
AC inputs during testing are driven at V_{CC} –0.5 for a logic '1' and 0.45V for a logic '0'.
Timing measurements are made at V_{IH} min for a logic '1' and V_{IL} max for a logic '0'.

SU00010

Figure 17. AC Testing Input/Output


V_{LOAD}+0.1V
V_{LOAD}
V_{LOAD}–0.1V
TIMING
REFERENCE
POINTS
V_{OH}–0.1V
V_{OL}+0.1V

NOTE:
For timing purposes, a port is no longer floating when a 100mV change from load voltage occurs, and begins to float when a 100mV change from the loaded V_{OH}/V_{OL} level occurs. $I_{OH}/I_{OL} \geq \pm 20mA$.

SU00011

Figure 18. Float Waveform

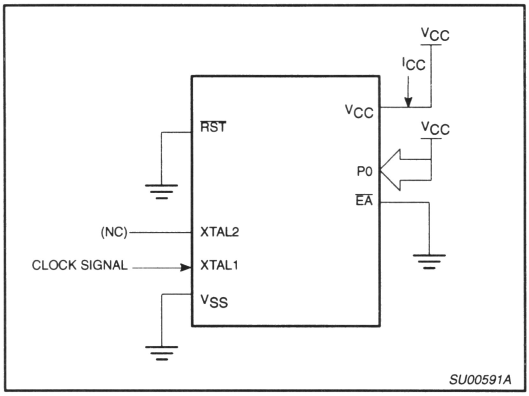

VCC
ICC
VCC
VCC
RST
P0
EA
(NC)
XTAL2
CLOCK SIGNAL
XTAL1
VSS
SU00591A

Figure 19. I_{CC} Test Condition, Active Mode
All other pins are disconnected

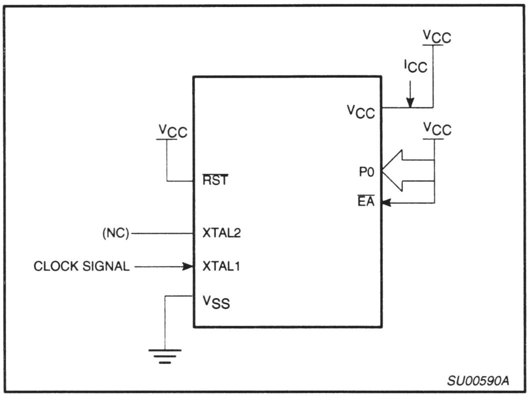

VCC
ICC
VCC
VCC
VCC
RST
P0
EA
(NC)
XTAL2
CLOCK SIGNAL
XTAL1
VSS
SU00590A

Figure 20. I_{CC} Test Condition, Idle Mode
All other pins are disconnected

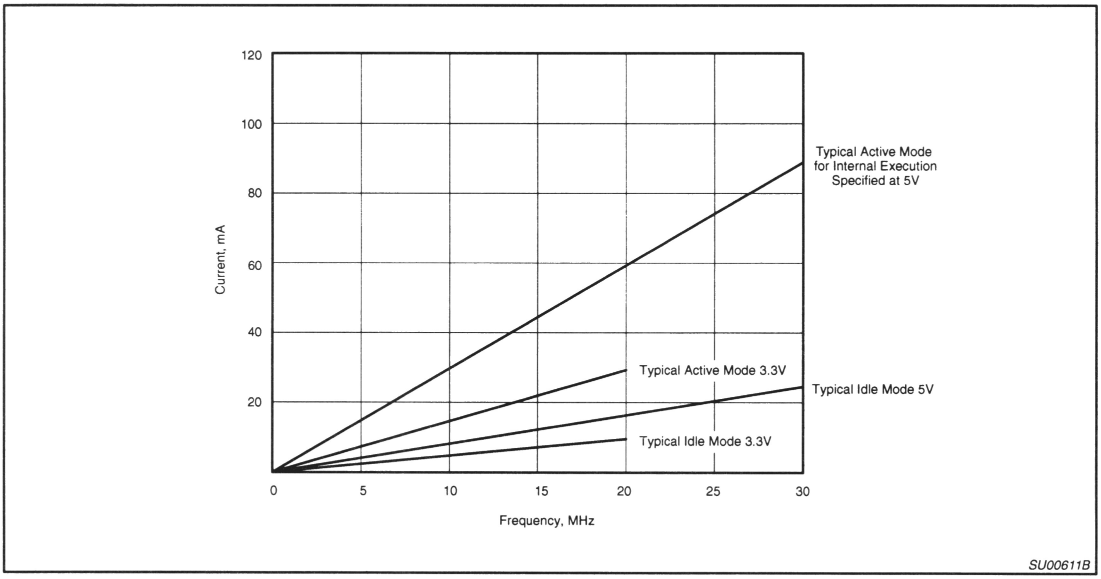


Figure 21. I_{CC} vs. Frequency
Valid only within frequency specification of the device under test.

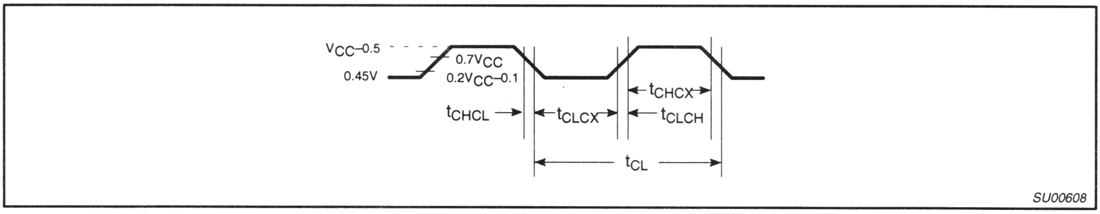

VCC–0.5
0.45V
0.7VCC
0.2VCC–0.1
tCHCL
tCLCX
tCHCX
tCLCH
tCL
SU00608

Figure 22. Clock Signal Waveform for I_{CC} Tests in Active and Idle Modes
$t_{CLCH} = t_{CHCL} = 5ns$

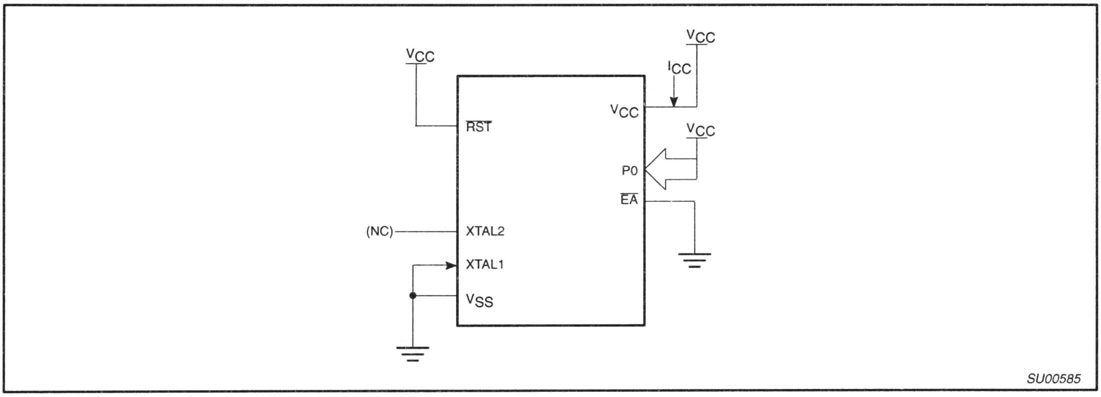


Figure 23. I_{CC} Test Condition, Power Down Mode
All other pins are disconnected. V_{CC}=2V to 5.5V

EPROM CHARACTERISTICS

The XA-G2 is programmed by using a modified Improved Quick-Pulse Programming™ algorithm. This algorithm is essentially the same as that used by the later 80C51 family EPROM parts. However different pins are used for many programming functions.

The XA-G2 contains three signature bytes that can be read and used by an EPROM programming system to identify the device. The signature bytes identify the device as an XA-G2 manufactured by Philips.

Table 5 shows the logic levels for reading the signature byte, and for programming the code memory and the security bits. The circuit configuration and waveforms for quick-pulse programming are shown in Figure 24. Figure 26 shows the circuit configuration for normal code memory verification.

Quick-Pulse Programming

The setup for microcontroller quick-pulse programming is shown in Figure 24. Note that the XA-G2 is running with a 3.5 to 12MHz oscillator. The reason the oscillator needs to be running is that the device is executing internal address and program data transfers.

The address of the EPROM location to be programmed is applied to ports 2 and 3, as shown in Figure 24. The code byte to be programmed into that location is applied to port 0. RST, $\overline{\text{PSEN}}$ and pins of port 1 specified in Table 5 are held at the 'Program Code Data' levels indicated in Table 5. The ALE/$\overline{\text{PROG}}$ is pulsed low 5 times as shown in Figure 25.

To program the security bits, repeat the 5 pulse programming sequence using the 'Pgm Security Bit' levels. After one security bit is programmed, further programming of the code memory and encryption table is disabled. However, the other security bits can still be programmed.

Note that the $\overline{\text{EA}}/V_{PP}$ pin must not be allowed to go above the maximum specified V_{PP} level for any amount of time. Even a narrow glitch above that voltage can cause permanent damage to the device. The V_{PP} source should be well regulated and free of glitches and overshoot.

Program Verification

If security bits 2 and 3 have not been programmed, the on-chip program memory can be read out for program verification. The address of the program memory locations to be read is applied to ports 2 and 3 as shown in Figure 26. The other pins are held at the 'Verify Code Data' levels indicated in Table 5. The contents of the address location will be emitted on port 0.

Reading the Signature Bytes

The signature bytes are read by the same procedure as a normal verification of locations 030H, 031H, and 060H except that P1.2 and P1.3 need to be pulled to a logic low. The values are:

(030H) = 15H indicates manufactured by Philips
(031H) = EAH indicates XA architecture
(060H) = 02H indicates XA-G2

Program/Verify Algorithms

Any algorithm in agreement with the conditions listed in Table 5, and which satisfies the timing specifications, is suitable.

Erasure Characteristics

Erasure of the EPROM begins to occur when the chip is exposed to light with wavelengths shorter than approximately 4,000 angstroms. Since sunlight and fluorescent lighting have wavelengths in this range, exposure to these light sources over an extended time (about 1 week in sunlight, or 3 years in room level fluorescent lighting) could cause inadvertent erasure. **For this and secondary effects, it is recommended that an opaque label be placed over the window.** For elevated temperature or environments where solvents are being used, apply Kapton tape Fluorglas part number 2345–5, or equivalent.

The recommended erasure procedure is exposure to ultraviolet light (at 2537 angstroms) to an integrated dose of at least 15W-s/cm^2. Exposing the EPROM to an ultraviolet lamp of 12,000µW/cm^2 rating for 90 to 120 minutes, at a distance of about 1 inch, should be sufficient.

Erasure leaves the array in an all 1s state.

™Trademark phrase of Intel Corporation.

Security Bits

With none of the security bits programmed the code in the program memory can be verified. When only security bit 1 (see Table 5) is programmed, MOVC instructions executed from external program memory are disabled from fetching code bytes from the internal memory. All further programming of the EPROM is disabled. When security bits 1 and 2 are programmed, in addition to the above, verify mode is disabled. When all three security bits are programmed, all of the conditions above apply and all external program memory execution is disabled. (See Table 6.)

Table 5. EPROM Programming Modes

MODE	$\overline{RST}$	$\overline{PSEN}$	ALE/$\overline{PROG}$	$\overline{EA}$/V_{PP}	P1.0	P1.1	P1.2	P1.3	P1.4
Read signature	0	0	1	1	0	0	0	0	0
Program code data	0	0	0*	V_{PP}	0	1	1	1	1
Verify code data	0	0	1	1	0	0	1	1	0
Pgm security bit 1	0	0	0*	V_{PP}	1	1	1	1	1
Pgm security bit 2	0	0	0*	V_{PP}	1	1	0	0	1
Pgm security bit 3	0	0	0*	V_{PP}	1	0	1	0	1
Verify security bits	0	0	1	1	0	0	0	1	0

NOTES:
1. ‘0’ = Valid low for that pin, ‘1’ = valid high for that pin.
2. V_{PP} = 12.75V ±0.25V.
3. V_{CC} = 5V±10% during programming and verification.

* ALE/$\overline{PROG}$ receives 5 programming pulses (only for user array; 25 pulses for encryption or security bits) while V_{PP} is held at 12.75V. Each programming pulse is low for 100µs (±10µs) and high for a minimum of 10µs.

Table 6. Program Security Bits

PROGRAM LOCK BITS				
	SB1	SB2	SB3	PROTECTION DESCRIPTION
1	U	U	U	No Program Security features enabled.
2	P	U	U	MOVC instructions executed from external program memory are disabled from fetching code bytes from internal memory and further programming of the EPROM is disabled.
3	P	P	U	Same as 2, also verify is disabled.
4	P	P	P	Same as 3, external execution is disabled. Internal data RAM is not accessible.

NOTES:
1. P – programmed. U – unprogrammed.
2. Any other combination of the security bits is not defined.

ROM CODE SUBMISSION

When submitting ROM code for the XA-G2, the following must be specified:
1. 16k byte user ROM data
2. ROM security bits.
3. Watchdog configuration

ADDRESS	CONTENT	BIT(S)	COMMENT
0000H to 3FFFH	DATA	7:0	User ROM Data
8020H	SEC	0	ROM Security Bit 1
8020H	SEC	1	ROM Security Bit 2 0 = enable security 1 = disable security
8020H	SEC	3	ROM Security Bit 3 0 = enable security 1 = disable security

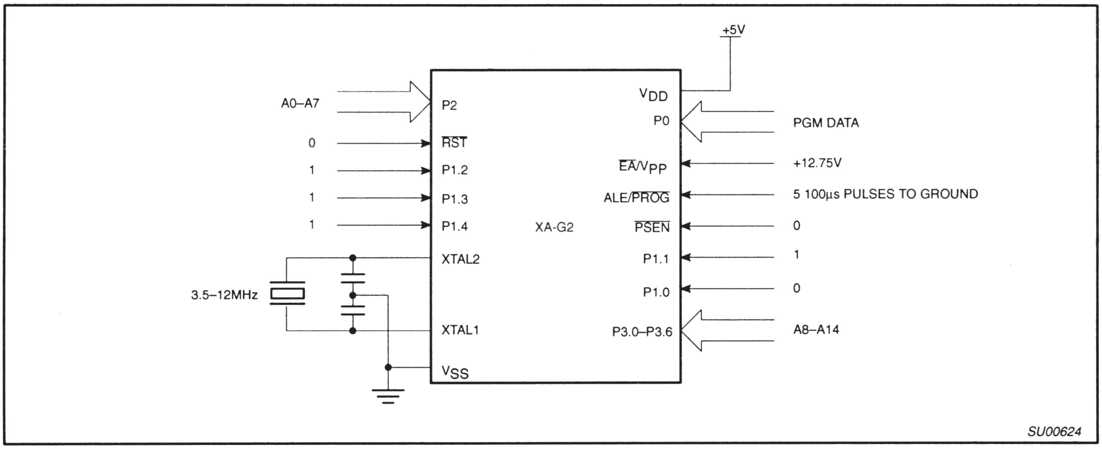

+5V
VDD
A0–A7
P2
P0
PGM DATA
0
RST
1
P1.2
EA/VPP
+12.75V
1
P1.3
ALE/PROG
5 100µs PULSES TO GROUND
1
P1.4
XA-G2
PSEN
0
XTAL2
P1.1
1
3.5–12MHz
P1.0
0
XTAL1
P3.0–P3.6
A8–A14
VSS
SU00624

Figure 24. Programming Configuration for XA-G2

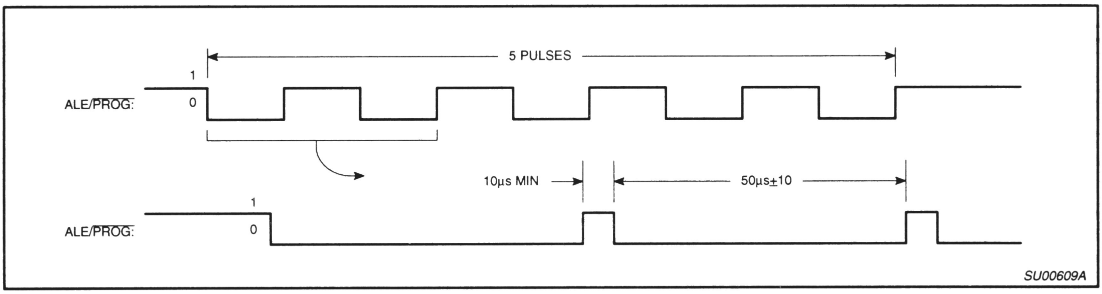

5 PULSES
1
ALE/PROG:
0
10µs MIN
50µs±10
1
ALE/PROG:
0
SU00609A

Figure 25. $\overline{\text{PROG}}$ Waveform

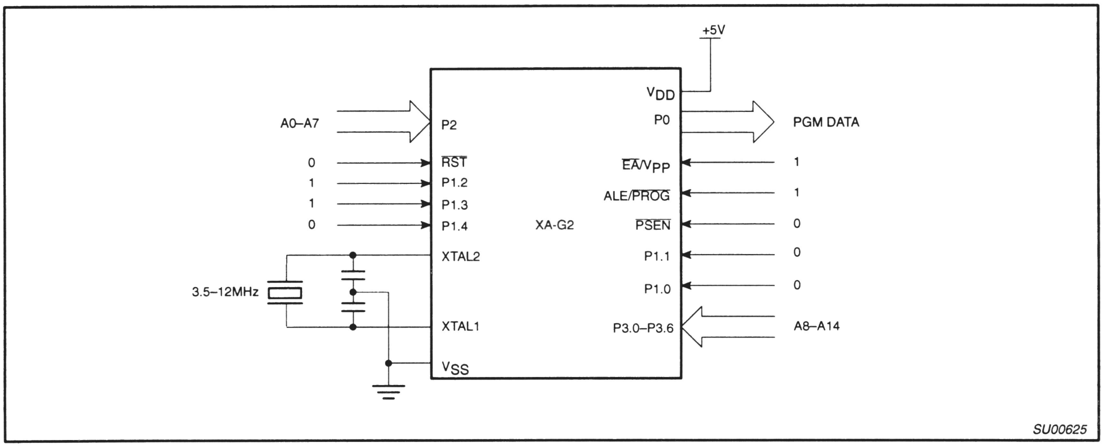

+5V
VDD
A0–A7
P2
P0
PGM DATA
0
RST
EA/VPP
1
1
P1.2
ALE/PROG
1
1
P1.3
0
P1.4
XA-G2
PSEN
0
XTAL2
P1.1
0
3.5–12MHz
P1.0
0
XTAL1
P3.0–P3.6
A8–A14
VSS
SU00625

Figure 26. Program Verification for XA-G2

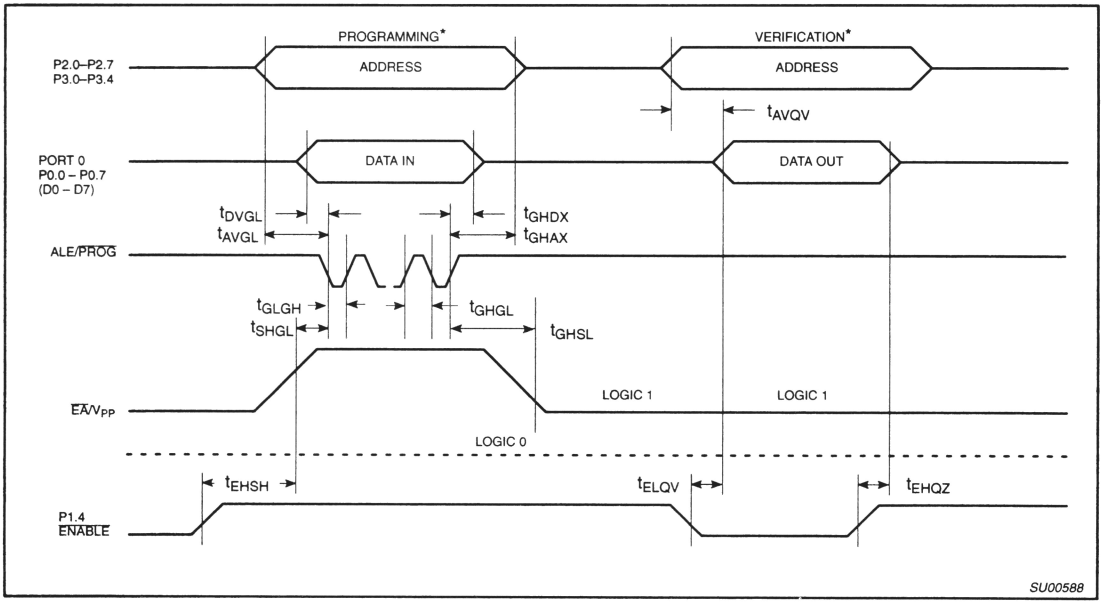
EPROM PROGRAMMING AND VERIFICATION CHARACTERISTICS

T_{amb} = 21°C to +27°C, V_{DD} = 5V±10%, V_{SS} = 0V (See Figure 27)

SYMBOL	PARAMETER	MIN	MAX	UNIT
V_{PP}	Programming supply voltage	12.5	13.0	V
I_{PP}	Programming supply current		50 [1]	mA
$1/t_{CL}$	Oscillator frequency	3.5	12	MHz
t_{AVGL}	Address setup to $\overline{PROG}$ low	$48t_{CL}$		
t_{GHAX}	Address hold after $\overline{PROG}$	$48t_{CL}$		
t_{DVGL}	Data setup to $\overline{PROG}$ low	$48t_{CL}$		
t_{GHDX}	Data hold after $\overline{PROG}$	$48t_{CL}$		
t_{EHSH}	P2.7 ($\overline{ENABLE}$) high to V_{PP}	$48t_{CL}$		
t_{SHGL}	V_{PP} setup to $\overline{PROG}$ low	10		µs
t_{GHSL}	V_{PP} hold after $\overline{PROG}$	10		µs
t_{GLGH}	$\overline{PROG}$ width	40	60	µs
t_{AVQV}	Address to data valid		$48t_{CL}$	
t_{ELQV}	$\overline{ENABLE}$ low to data valid		$48t_{CL}$	
t_{EHQZ}	Data float after $\overline{ENABLE}$	0	$48t_{CL}$	
t_{GHGL}	$\overline{PROG}$ high to $\overline{PROG}$ low	10		µs

NOTE:

1. Not tested.


PROGRAMMING*
VERIFICATION*
P2.0–P2.7
P3.0–P3.4
ADDRESS
ADDRESS
tAVQV
PORT 0
P0.0 – P0.7
(D0 – D7)
DATA IN
DATA OUT
tDVGL
tAVGL
tGHDX
tGHAX
ALE/PROG
tGLGH
tSHGL
tGHGL
tGHSL
EA/VPP
LOGIC 1
LOGIC 1
LOGIC 0
tEHSH
tELQV
tEHQZ
P1.4
ENABLE
SU00588

NOTE:

* FOR PROGRAMMING CONDITIONS SEE FIGURE 25.
FOR VERIFICATION CONDITIONS SEE FIGURE 26.

Figure 27. EPROM Programming and Verification

CMOS single-chip 16-bit microcontroller XA-G3

FAMILY DESCRIPTION

The Philips Semiconductors XA (eXtended Architecture) family of 16-bit single-chip microcontrollers is powerful enough to easily handle the requirements of high performance embedded applications, yet inexpensive enough to compete in the market for high-volume, low-cost applications.

The XA family provides an upward compatibility path for 80C51 users who need higher performance and 64k or more of program memory. Existing 80C51 code can also be easily be translated to run on XA microcontrollers.

The performance of the XA architecture supports the comprehensive bit-oriented operations of the 80C51 while incorporating support for multi-tasking operating systems and high-level languages such as C. The speed of the XA architecture, at 10 to 100 times that of the 80C51, gives designers and easy path to truly high performance embedded control.

The XA architecture supports:

- Upward compatibility with the 80C51 architecture
- 16-bit fully static CPU with a 24-bit program and data address range
- Eight 16-bit CPU registers each capable of performing all arithmetic and logic operations as well as acting as memory pointers. Operations may also be performed directly to memory.
- Both 8-bit and 16-bit CPU registers, each capable of performing all arithmetic and logic operations.
- An enhanced instruction set that includes bit intensive logic operations and fast signed or unsigned 16 × 16 multiply and 32 / 16 divide
- Instruction set tailored for high level language support
- Multi-tasking and real-time executives that include up to 32 vectored interrupts, 16 software traps, segmented data memory, and banked registers to support context switching
- Low power operation, which is intrinsic to the XA architecture, includes power-down and idle modes.

More detailed information on the core is available in the XA User Guide.

SPECIFIC FEATURES OF THE XA-G3

- 20-bit address range, 1 megabyte each program and data space. (Note that the XA architecture supports up to 24 bit addresses.)
- 2.7V to 5.5V operation
- 32K bytes on-chip EPROM/ROM program memory
- 512 bytes of on-chip data RAM
- Three counter/timers with enhanced features (equivalent to 80C51 T0, T1, and T2)
- Watchdog timer
- Two enhanced UARTs
- Four 8-bit I/O ports with 4 programmable output configurations
- 44-pin PLCC and 44-pin LQFP packages

ORDERING INFORMATION

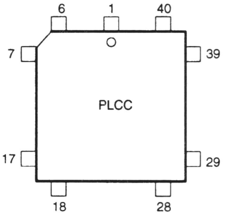
ROMless	ROM	EPROM[1]		TEMPERATURE RANGE °C AND PACKAGE	FREQ (MHz)	DRAWING NUMBER
P51XAG30KB BD	P51XAG33KB BD	P51XAG37KB BD	OTP	0 to +70, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG30KB A	P51XAG33KB A	P51XAG37KB A	OTP	0 to +70, Plastic Leaded Chip Carrier	30	SOT187–2
		P51XAG37KB KA	UV	0 to +70, Ceramic Leaded Chip Carrier	30	1472A
P51XAG30KF BD	P51XAG33KF BD	P51XAG37KF BD	OTP	–40 to +85, Plastic Low Profile Quad Flat Pkg.	30	SOT389–1
P51XAG30KF A	P51XAG33KF A	P51XAG37KF A	OTP	–40 to +85, Plastic Leaded Chip Carrier	30	SOT187–2
		P51XAG37KF KA	UV	–40 to +85, Ceramic Leaded Chip Carrier	30	1472A

NOTE:

1. OTP = One Time Programmable EPROM. UV = Erasable EPROM.

PIN CONFIGURATIONS

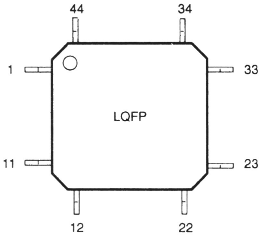
44-Pin PLCC Package



Pin	Function	Pin	Function
1	V_{SS}	23	V_{DD}
2	P1.0/A0/$\overline{WRH}$	24	P2.0/A12D8
3	P1.1/A1	25	P2.1/A13D9
4	P1.2/A2	26	P2.2/A14D10
5	P1.3/A3	27	P2.3/A15D11
6	P1.4/RxD1	28	P2.4/A16D12
7	P1.5/TxD1	29	P2.5/A17D13
8	P1.6/T2	30	P2.6/A18D14
9	P1.7/T2EX	31	P2.7/A19D15
10	$\overline{RST}$	32	$\overline{PSEN}$
11	P3.0/RxD0	33	ALE/$\overline{PROG}$
12	NC	34	NC
13	P3.1/TxD0	35	$\overline{EA}$/V_{PP}/WAIT
14	P3.2/$\overline{INT0}$	36	P0.7/A11D7
15	P3.3/$\overline{INT1}$	37	P0.6/A10D6
16	P3.4/T0	38	P0.5/A9D5
17	P3.5/T1/BUSW	39	P0.4/A8D4
18	P3.6/$\overline{WRL}$	40	P0.3/A7D3
19	P3.7/$\overline{RD}$	41	P0.2/A6D2
20	XTAL2	42	P0.1/A5D1
21	XTAL1	43	P0.0/A4D0
22	V_{SS}	44	V_{DD}

SU00525

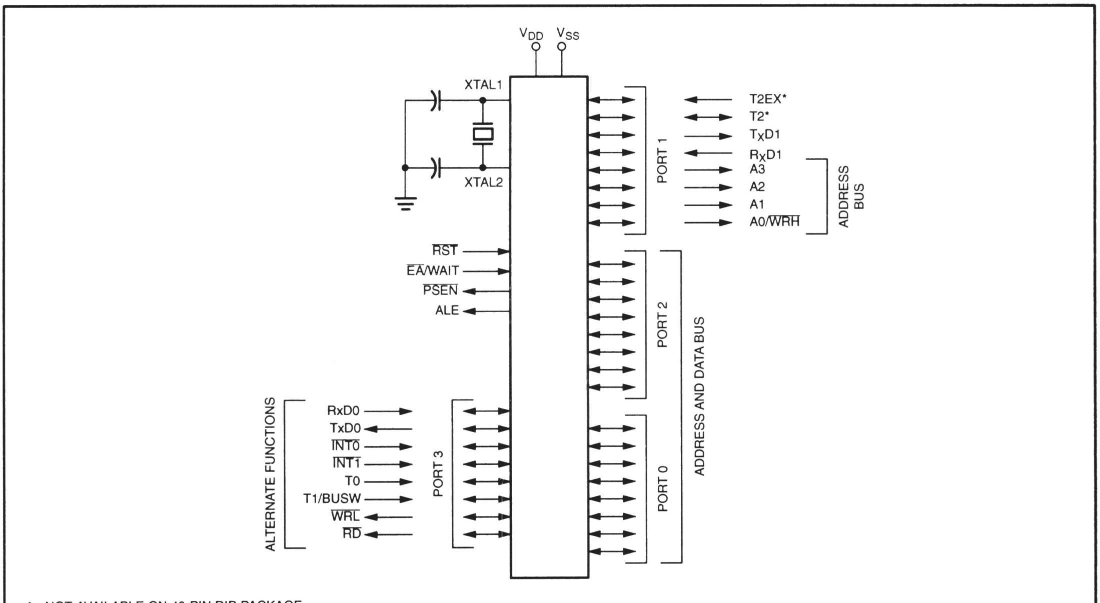
44-Pin LQFP Package



Pin	Function	Pin	Function
1	P1.5/TxD1	23	P2.5/A17D13
2	P1.6/T2	24	P2.6/A18D14
3	P1.7/T2EX	25	P2.7/A19D15
4	$\overline{RST}$	26	$\overline{PSEN}$
5	P3.0/RxD0	27	ALE/$\overline{PROG}$
6	NC	28	NC
7	P3.1/TxD0	29	$\overline{EA}$/V_{PP}/WAIT
8	P3.2/$\overline{INT0}$	30	P0.7/A11D7
9	P3.3/$\overline{INT1}$	31	P0.6/A10D6
10	P3.4/T0	32	P0.5/A9D5
11	P3.5/T1/BUSW	33	P0.4/A8D4
12	P3.6/$\overline{WRL}$	34	P0.3/A7D3
13	P3.7/$\overline{RD}$	35	P0.2/A6D2
14	XTAL2	36	P0.1/A5D1
15	XTAL1	37	P0.0/A4D0
16	V_{SS}	38	V_{DD}
17	V_{DD}	39	V_{SS}
18	P2.0/A12D8	40	P1.0/A0/$\overline{WRH}$
19	P2.1/A13D9	41	P1.1/A1
20	P2.2/A14D10	42	P1.2/A2
21	P2.3/A15D11	43	P1.3/A3
22	P2.4/A16/D12	44	P1.4/RxD1

SU00580

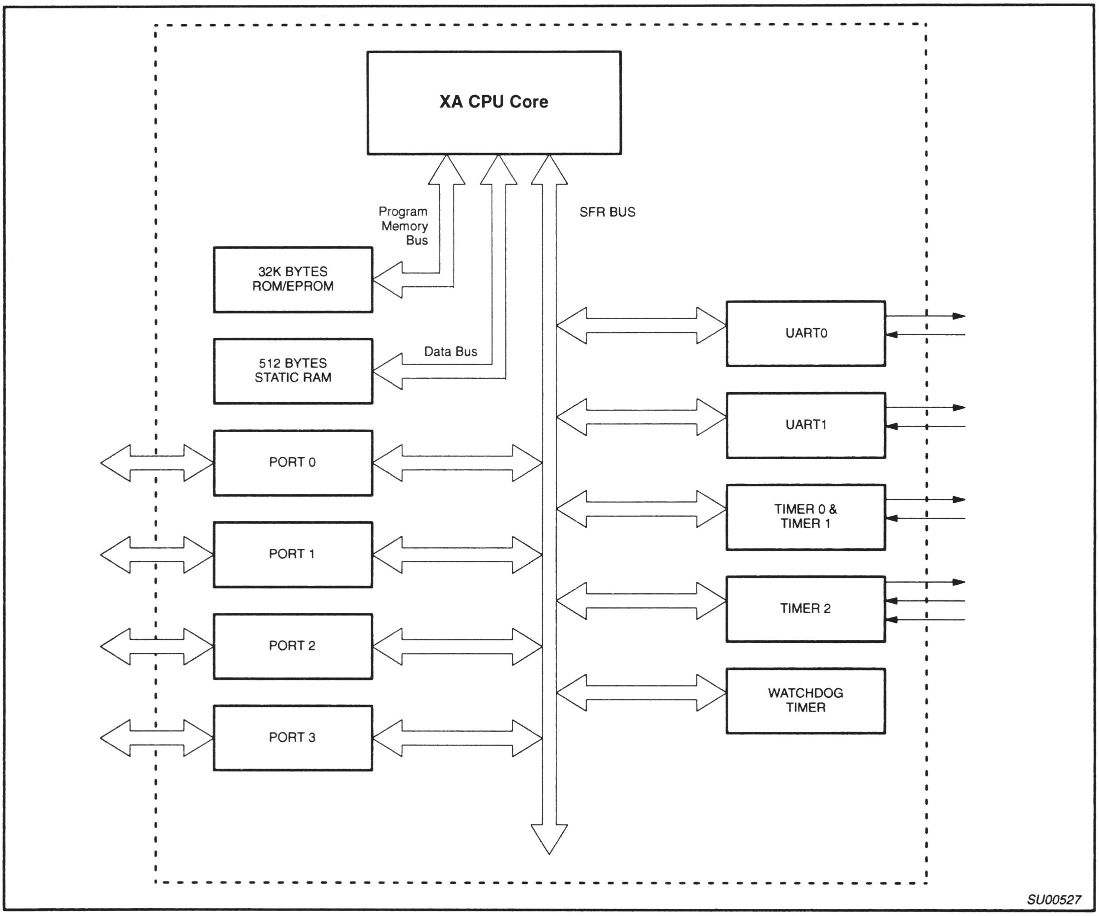
LOGIC SYMBOL



V_{DD}
V_{SS}
XTAL1
XTAL2
RST
EA/WAIT
PSEN
ALE
PORT 1
T2EX*
T2*
T_XD1
R_XD1
A3
A2
A1
A0/WRH
ADDRESS BUS
PORT 2
ADDRESS AND DATA BUS
PORT 0
PORT 3
ALTERNATE FUNCTIONS
RxD0
TxD0
INT0
INT1
T0
T1/BUSW
WRL
RD
SU00526

* NOT AVAILABLE ON 40-PIN DIP PACKAGE

BLOCK DIAGRAM


XA CPU Core
Program Memory Bus
SFR BUS
32K BYTES ROM/EPROM
512 BYTES STATIC RAM
Data Bus
PORT 0
PORT 1
PORT 2
PORT 3
UART0
UART1
TIMER 0 & TIMER 1
TIMER 2
WATCHDOG TIMER
SU00527

PIN DESCRIPTIONS

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
V_{SS}	1, 22	16	I	**Ground:** 0V reference.
V_{DD}	23, 44	17	I	**Power Supply:** This is the power supply voltage for normal, idle, and power down operation.
P0.0 – P0.7	43–36	37–30	I/O	**Port 0:** Port 0 is an 8-bit I/O port with a user-configurable output type. Port 0 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 0 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used, Port 0 becomes the multiplexed low data/instruction byte and address lines 4 through 11. Port 0 also outputs the code bytes during program verification and receives code bytes during EPROM programming.
P1.0 – P1.7	2–9	40–44, 1–3	I/O	**Port 1:** Port 1 is an 8-bit I/O port with a user-configurable output type. Port 1 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 1 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 1 also provides special functions as described below.
	2	40	O	**A0/$\overline{\mathrm{WRH}}$:** Address bit 0 of the external address bus when the external data bus is configured for an 8 bit width. When the external data bus is configured for a 16 bit width, this pin becomes the high byte write strobe.
	3	41	O	**A1:** Address bit 1 of the external address bus.
	4	42	O	**A2:** Address bit 2 of the external address bus.
	5	43	O	**A3:** Address bit 3 of the external address bus.
				Port 1 also provides various special functions as described below.
	6	44	I	**RxD1 (P1.4):** Receiver input for serial port 1.
	7	1	O	**TxD1 (P1.5):** Transmitter output for serial port 1.
	8	2	I	**T2 (P1.6):** Timer/counter 2 external count input/clockout.
	9	3	I	**T2EX (P1.7):** Timer/counter 2 reload/capture/direction control
P2.0 – P2.7	24–31	18–25	I/O	**Port 2:** Port 2 is an 8-bit I/O port with a user-configurable output type. Port 2 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. The operation of port 2 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. When the external program/data bus is used in 16-bit mode, Port 2 becomes the multiplexed high data/instruction byte and address lines 12 through 19. When the external program/data bus is used in 8-bit mode, the number of address lines that appear on port 2 is user programmable. Port 2 also receives the low-order address byte during program memory verification.
P3.0 – P3.7	11, 13–19	5, 7–13	I/O	**Port 3:** Port 3 is an 8-bit I/O port with a user configurable output type. Port 3 latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. the operation of port 3 pins as inputs and outputs depends upon the port configuration selected. Each port pin is configured independently. Refer to the section on I/O port configuration and the DC Electrical Characteristics for details. Port 3 pins receive the high order address bits during EPROM programming and verification. Port 3 also provides various special functions as described below.
	11	5	I	**RxD0 (P3.0):** Receiver input for serial port 0.
	13	7	O	**TxD0 (P3.1):** Transmitter output for serial port 0.
	14	8	I	**$\overline{\mathrm{INT0}}$ (P3.2):** External interrupt 0 input.
	15	9	I	**$\overline{\mathrm{INT1}}$ (P3.3):** External interrupt 1 input.
	16	10	I/O	**T0 (P3.4):** Timer 0 external input, or timer 0 overflow output.
	17	11	I/O	**T1/BUSW (P3.5):** Timer 1 external input, or timer 1 overflow output. The value on this pin is latched as the external reset input is released and defines the default external data bus width (BUSW).
	18	12	O	**$\overline{\mathrm{WRL}}$ (P3.6):** External data memory low byte write strobe.
	19	13	O	**$\overline{\mathrm{RD}}$ (P3.7):** External data memory read strobe.

MNEMONIC	PIN. NO.		TYPE	NAME AND FUNCTION
	LCC	LQFP		
$\overline{RST}$	10	4	I	**Reset:** A low on this pin resets the microcontroller, causing I/O ports and peripherals to take on their default states, and the processor to begin execution at the address contained in the reset vector. Refer to the section on Reset for details.
ALE/$\overline{PROG}$	33	27	I/O	**Address Latch Enable/Program Pulse:** A high output on the ALE pin signals external circuitry to latch the address portion of the multiplexed address/data bus. A pulse on ALE occurs only when it is needed in order to process a bus cycle. During EPROM programming, this pin is used as the program pulse input.
$\overline{PSEN}$	32	26	O	**Program Store Enable:** The read strobe for external program memory. When the microcontroller accesses external program memory, $\overline{PSEN}$ is driven low in order to enable memory devices. $\overline{PSEN}$ is only active when external code accesses are performed.
$\overline{EA}$/WAIT/ V_{PP}	35	29	I	**External Access/Wait/Programming Supply Voltage:** The $\overline{EA}$ input determines whether the internal program memory of the microcontroller is used for code execution. The value on the $\overline{EA}$ pin is latched as the external reset input is released and applies during later execution. When latched as a 0, external program memory is used exclusively, when latched as a 1, internal program memory will be used up to its limit, and external program memory used above that point. After reset is released, this pin takes on the function of bus Wait input. If Wait is asserted high during any external bus access, that cycle will be extended until Wait is released. During EPROM programming, this pin is also the programming supply voltage input.
XTAL1	21	15	I	**Crystal 1:** Input to the inverting amplifier used in the oscillator circuit and input to the internal clock generator circuits.
XTAL2	20	14	O	**Crystal 2:** Output from the oscillator amplifier.

SPECIAL FUNCTION REGISTERS

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
BCR	Bus configuration register	46A	—	—	—	WAITD	BUSD	BC2	BC1	BC0	Note 1
BTRH	Bus timing register low byte	469	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0	FF
BTRL	Bus timing register high byte	468	WM1	WM0	ALEW	—	CR1	CR0	CRA1	CRA0	EF
CS	Code segment	443									00
DS	Data segment	441									00
ES	extra segment	442									00
			33F	33E	33D	33C	33B	33A	339	338	
IEH*	Interrupt enable high byte	427	—	—	—	—	ETI1	ERI1	ETI0	ERI0	00
			337	336	335	334	333	332	331	330	
IEL*	Interrupt enable low byte	426	EA	—	—	ET2	ET1	EX1	ET0	EX0	00
IPA0	Interrupt priority 0	4A0	—	PT0			—	PX0			00
IPA1	Interrupt priority 1	4A1	—	PT1			—	PX1			00
IPA2	Interrupt priority 2	4A2	—	—			—	PT2			00
IPA4	Interrupt priority 4	4A4	—	PTI0			—	PRI0			00
IPA5	Interrupt priority 5	4A5	—	PTI1			—	PRI1			00
			387	386	385	384	383	382	381	380	
P0*	Port 0	430	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FF
			38F	38E	38D	38C	38B	38A	389	388	
P1*	Port 1	431	T2EX	T2	P1.5	P1.4	P1.3	P1.2	P1.1	WR1	FF
			397	396	395	394	393	392	391	390	
P2*	Port 2	432	P2.7	P2.6	P2.5	P2.4	TxD1	RxD1	P2.1	P2.0	FF

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
			39F	39E	39D	39C	39B	39A	399	398	
P3*	Port 3	433	RD	WR	T1	T0	INT1	INT0	TxD0	RxD0	FF
P0CFGA	Port 0 configuration A	470									Note 5
P1CFGA	Port 1 configuration A	471									Note 5
P2CFGA	Port 2 configuration A	472									Note 5
P3CFGA	Port 3 configuration A	473									Note 5
P0CFGB	Port 0 configuration B	4F0									Note 5
P1CFGB	Port 1 configuration B	4F1									Note 5
P2CFGB	Port 2 configuration B	4F2									Note 5
P3CFGB	Port 3 configuration B	4F3									Note 5
			227	226	225	224	223	222	221	220	
PCON*	Power control register	404	—	—	—	—	—	—	PD	IDL	00
			20F	20E	20D	20C	20B	20A	209	208	
PSWH*	Program status word (high byte)	401	SM	TM	RS1	RS0	IM3	IM2	IM1	IM0	Note 2
			207	206	205	204	203	202	201	200	
PSWL*	Program status word (low byte)	400	C	AC	—	—	—	V	N	Z	Note 2
			217	216	215	214	213	212	211	210	
PSW51*	80C51 compatible PSW	402	C	AC	F0	RS1	RS0	V	F1	P	Note 3
RTH0	Timer 0 extended reload, high byte	455									00
RTH1	Timer 1 extended reload, high byte	457									00
RTL0	Timer 0 extended reload, low byte	454									00
RTL1	Timer 1 extended reload, low byte	456									00
			307	306	305	304	303	302	301	300	
S0CON*	Serial port 0 control register	420	SM0_0	SM1_0	SM2_0	REN_0	TB8_0	RB8_0	TI_0	RI_0	00
			30F	30E	30D	30C	30B	30A	309	308	
S0STAT*	Serial port 0 extended status	421	—	—	—	—	FE0	BR0	OE0	STINT0	00
S0BUF	Serial port 0 buffer register	460									x
S0ADDR	Serial port 0 address register	461									00
S0ADEN	Serial port 0 address enable register	462									00
			327	326	325	324	323	322	321	320	
S1CON*	Serial port 1 control register	424	SM0_1	SM1_1	SM2_1	REN_1	TB8_1	RB8_1	TI_1	RI_1	00
			32F	32E	32D	32C	32B	32A	329	328	
S1STAT*	Serial port 1 extended status	425	—	—	—	—	FE1	BR1	OE1	STINT1	00
S1BUF	Serial port 1 buffer register	464									x
S1ADDR	Serial port 1 address register	465									00
S1ADEN	Serial port 1 address enable register	466									00
SCR	System configuration register	440	—	—	—	—	PT1	PT0	CM	PZ	00
			21F	21E	21D	21C	21B	21A	219	218	
SSEL*	Segment selection register	403	ESWEN	R6SEG	R5SEG	R4SEG	R3SEG	R2SEG	R1SEG	R0SEG	00
SWE	Software Interrupt Enable	47A	—	SWE7	SWE6	SWE5	SWE4	SWE3	SWE2	SWE1	00

NAME	DESCRIPTION	SFR ADDRESS	BIT FUNCTIONS AND ADDRESSES								RESET VALUE
			MSB							LSB	
			357	356	355	354	353	352	351	350	
SWR*	Software Interrupt Request	42A	—	SWR7	SWR6	SWR5	SWR4	SWR3	SWR2	SWR1	00
			2C7	2C6	2C5	2C4	2C3	2C2	2C1	2C0	
T2CON*	Timer 2 control register	418	TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2	00
			2CF	2CE	2CD	2CC	2CB	2CA	2C9	2C8	
T2MOD*	Timer 2 mode control	419	—	—	RCLK1	TCLK1	—	T2RD	T2OE	DCEN	00
TH2	Timer 2 high byte	459									00
TL2	Timer 2 low byte	458									00
T2CAPH	Timer 2 capture register, high byte	45B									00
T2CAPL	Timer 2 capture register, low byte	45A									00
			287	286	285	284	283	282	281	280	
TCON*	Timer 0 and 1 control register	410	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00
TH0	Timer 0 high byte	451									00
TH1	Timer 1 high byte	453									00
TL0	Timer 0 low byte	450									00
TL1	Timer 1 low byte	452									00
TMOD	Timer 0 and 1 mode register	45C	GATE	C/T	M1	M0	GATE	C/T	M1	M0	00
			28F	28E	28D	28C	28B	28A	289	288	
TSTAT*	Timer 0 and 1 extended status	411	—	—	—	—	T1RD	T1OE	T0RD	T0OE	00
			2FF	2FE	2FD	2FC	2FB	2FA	2F9	2F8	
WDCON*	Watchdog control register	41F	PRE2	PRE1	PRE0	—	—	WDRUN	WDTOF	—	Note 6
WDL	Watchdog timer reload	45F									00
WFEED1	Watchdog feed 1	45D									x
WFEED2	Watchdog feed 2	45E									x

NOTES:

* SFRs are bit addressable.

1. At reset, the BCR register is loaded with the binary value 0000 0a11, where “a” is the value on the BUSW pin.
2. SFR is loaded from the reset vector.
3. All bits except F1, F0, and P are loaded from the reset vector. Those bits are all 0.
4. Unimplemented bits in SFRs are X (unknown) at all times. Ones should not be written to these bits since they may be used for other purposes in future XA derivatives. The reset value shown for these bits is 0.
5. Port configurations default to quasi-bidirectional when the XA begins execution from internal code memory after reset, based on the condition found on the EA pin. Thus all PnCFGA registers will contain FF and PnCFGB registers will contain 00. When the XA begins execution using external code memory, the default configuration for pins that are associated with the external bus will be push-pull. The PnCFGA and PnCFGB register contents will reflect this difference.
6. The WDCON reset value is E6 for a Watchdog reset, E4 for all other reset causes.

XA-G3 TIMER/COUNTERS

The XA has two standard 16-bit enhanced Timer/Counters: Timer 0 and Timer 1. Additionally, it has a third 16-bit Up/Down timer/counter, T2. A central timing generator in the XA core provides the time-base for all XA Timers and Counters. The timer/event counters can perform the following functions:

- Measure time intervals and pulse duration
- Count external events
- Generate interrupt requests
- Generate PWM or timed output waveforms

All of the XA-G3 timer/counters (Timer 0, Timer 1 and Timer 2) can be independently programmed to operate either as timers or event counters via the C/T bit in the TnCON register. These timers may be dynamically read during program execution.

The base clock rate of all of the XA-G3 timers is user programmable. This applies to timers T0, T1, and T2 when running in timer mode (as opposed to counter mode), and the watchdog timer. The clock driving the timers is called TCLK and is determined by the setting of two bits (PT1, PT0) in the System Configuration Register (SCR). The frequency of TCLK may be selected to be the oscillator input divided by 4 (Osc/4), the oscillator input divided by 16 (Osc/16), or the oscillator input divided by 64 (Osc/64). This gives a range of possibilities for the XA timer functions, including baud rate generation, Timer 2 capture. Note that this single rate setting applies to all of the timers.

When timers T0, T1, or T2 are used in the counter mode, the register will increment whenever a falling edge (high to low transition) is detected on the external input pin corresponding to the timer clock. These inputs are sampled once every 2 oscillator cycles, so it can take as many as 4 oscillator cycles to detect a transition. Thus the maximum count rate that can be supported is Osc/4. The duty cycle of the timer clock inputs is not important, but any high or low state on the timer clock input pins must be present for 2 oscillator cycles before it is guaranteed to be "seen" by the timer logic.

Timer 0 and Timer 1

The "Timer" or "Counter" function is selected by control bits C/T in the special function register TMOD. These two Timer/Counters have four operating modes, which are selected by bit-pairs (M1, M0) in the TMOD register. Timer modes 1, 2, and 3 in XA are kept identical to the 80C51 timer modes for code compatibility. Only the mode 0 is replaced in the XA by a more powerful 16-bit auto-reload mode. This will give the XA timers a much larger range when used as time bases.

The recommended M1, M0 settings for the different modes are shown in Figure 2.

SCR Address:440
Not Bit Addressable
Reset Value: 00H

MSB LSB

—	—	—	—	PT1	PT0	CM	PZ

PT1	PT0	OPERATING
		Prescaler selection.
0	0	Osc/4
0	1	Osc/16
1	0	Osc/64
1	1	Reserved
CM		Compatibility Mode allows the XA to execute most translated 80C51 code on the XA. The XA register file must copy the 80C51 mapping to data memory and mimic the 80C51 indirect addressing scheme.
PZ		Page Zero mode forces all program and data addresses to 16-bits only. This saves stack space and speeds up execution but limits memory access to 64k.

SU00589

Figure 1. System Configuration Register (SCR)

TMOD Address:45C
Not Bit Addressable
Reset Value: 00H

MSB LSB

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
TIMER 1				TIMER 0			

GATE	Gating control when set. Timer/Counter "n" is enabled only while "$\overline{\text{INTn}}$" pin is high and "TRn" control bit is set. When cleared Timer "n" is enabled whenever "TRn" control bit is set.
C/$\overline{T}$	Timer or Counter Selector cleared for Timer operation (input from internal system clock.) Set for Counter operation (input from "Tn" input pin).

M1	**M0**	**OPERATING**
0	0	16-bit auto-reload timer/counter
0	1	16-bit non-auto-reload timer/counter
1	0	8-bit auto-reload timer/counter
1	1	Dual 8-bit timer mode (timer 0 only)

SU00605

Figure 2. Timer/Counter Mode Control (TMOD) Register

New Enhanced Mode 0

For timers T0 or T1 the 13-bit count mode on the 80C51 (current Mode 0) has been replaced in the XA with a 16-bit auto-reload mode. Four additional 8-bit data registers (two per timer: RTHn and RTLn) are created to hold the auto-reload values. In this mode, the TH overflow will set the TF flag in the T2CON register and cause both the TL and TH counters to be loaded from the RTL and RTH registers respectively.

These new SFRs will also be used to hold the TL reload data in the 8-bit auto-reload mode (Mode 2) instead of TH.

Mode 1

Mode 1 is the 16-bit non-auto reload mode.

Mode 2

Mode 2 configures the Timer register as an 8-bit Counter (TLn) with automatic reload. Overflow from TLn not only sets TFn, but also reloads TLn with the contents of RTLn, which is preset by software. The reload leaves THn unchanged.

Mode 2 operation is the same for Timer/Counter 0.

Mode 3

Timer 1 in Mode 3 simply holds its count. The effect is the same as setting TR1 = 0.

Timer 0 in Mode 3 establishes TL0 and TH0 as two separate counters. TL0 uses the Timer 0 control bits: C/T, GATE, TR0, INT0, and TF0. TH0 is locked into a timer function and takes over the use of TR1 and TF1 from Timer 1. Thus, TH0 now controls the "Timer 1" interrupt.

Mode 3 is provided for applications requiring an extra 8-bit timer. When Timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt.

TCON Address:441
Bit Addressable
Reset Value: 00H

MSB LSB

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

BIT	**SYMBOL**	**FUNCTION**
TCON.7	TF1	Timer 1 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.6	TR1	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TCON.5	TF0	Timer 0 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or by clearing the bit in software.
TCON.4	TR0	Timer 0 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TCON.3	IE1	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.2	IT1	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
TCON.1	IE0	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TCON.0	IT0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

SU00604

Figure 3. Timer/Counter Control (TCON) Register

T2CON Address:418
Bit Addressable
Reset Value: 00H

MSB							LSB
TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2

BIT	SYMBOL	FUNCTION
T2CON.7	TF2	Timer 2 overflow flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine, or clearing the bit in software. TF2 will not be set when RCLK0, RCLK1, TCLK0, TCLK1 or T2OE=1.
T2CON.6	EXF2	Timer 2 external flag is set when a capture or reload occurs due to a negative transition on T2EX (and EXEN is set). This flag will cause a Timer 2 interrupt when this interrupt is enabled. EXF2 is cleared by software.
T2CON.5	RCLK0	Receive Clock Flag.
T2CON.4	TCLK0	Transmit Clock Flag. RCLK0 and TCLK0 are used to select Timer 2 overflow rate as a clock source for UART0.
T2CON.3	EXEN2	Timer 2 external enable flag allows a capture or reload to occure due to a negative transition on T2EX.
T2CON.2	TR2	Start=1/Stop=0 control for Timer 2.
T2CON.1	C/T2	Timer or counter select. 0=Internal timer 1=External event counter (falling edge triggered)
T2CON.0	IT0	Capture/Reload flag. If CP/RL2 & EXEN2=1 captures will occur on negative transitions of T2EX. If CP/RL2=0, EXEN2=1 auto reloads occur with either Timer 2 overflows or negative transitions at T2EX. If RCLK or TCLK=1 the timer is set to auto reload on Timer 2 overflow, this bit has no effect.

SU00606

Figure 4. Timer/Counter 2 Control (T2CON) Register

New Timer-Overflow Toggle Output

In the XA, the timer module now has two outputs, which toggle on overflow from the individual timers. The same device pins that are used for the T0 and T1 count inputs are also used for the new overflow outputs. An SFR bit (TnOE in the TSTAT register) is associated with each counter and indicates whether Port-SFR data or the overflow signal is output to the pin. These outputs could be used in applications for generating variable duty cycle PWM outputs (changing the auto-reload register values). Also variable frequency (Osc/8 to Osc/8,388,608) outputs could be achieved by adjusting the prescaler along with the auto-reload register values. With a 30.0MHz oscillator, this range would be 3.58Hz to 3.75MHz.

Timer T2

This is a 16-bit up or down counter, which can be operated as either a timer or event counter. It can be operated in one of three different modes (autoreload, capture or as the baud rate generator for either or both UARTs).

In the autoreload mode the Timer can be set to count up or down by setting or clearing the bit DCEN in the T2MOD Special Function Register. The SFR's T2CAPH and T2CAPL are used to reload the Timer upon overflow or to capture a 1-to-0 transition on the T2EX input (P1.7).

In the Capture mode Timer 2 can either set TF2 and generate an interrupt or capture its value. To capture Timer 2 in response to a 1-to-0 transition on the T2EX input, the EXEN2 bit in the T2CON must be set. Timer 2 is then captured in SFR's T2CAP2H and T2CAP2L.

As the baud rate generator, Timer 2 is selected by setting one of the RCLK and/or TCLK bits in T2CON or T2MOD. As the baud rate generator Timer 2 is incremented by TCLK.

Programmable Clock-Out

A 50% duty cycle clock can be programmed to come out on P1.6. This pin, besides being a regular I/O pin, has two alternate functions. It can be programmed (1) to input the external clock for Timer/Counter 2 or (2) to output a 50% duty cycle clock ranging from 3.58MHz to 3.75MHz at a 30MHz operating frequency.

To configure the Timer/Counter 2 as a clock generator, bit C/T2 (in T2CON) must be cleared and bit T20E in T2MOD must be set. Bit TR2 (T2CON.2) also must be set to start the timer.

The Clock-Out frequency depends on the oscillator frequency and the reload value of Timer 2 capture registers (TCAP2H, TCAP2L) as shown in this equation:

$$\frac{\text{TCLK}}{2 \times (65536 - \text{TCAP2H, TCAP2L})}$$

In the Clock-Out mode Timer 2 roll-overs will not generate an interrupt. This is similar to when it is used as a baud-rate generator. It is possible to use Timer 2 as a baud-rate generator and a clock generator simultaneously. Note, however, that the baud-rate and the Clock-Out frequency will be the same.

TSTAT Address:411
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	—	—	T1RD	T1OE	T0RD	T0OE

BIT	SYMBOL	FUNCTION
TSTAT.3	T1RD	This bit is set, for Timer 1, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
TSTAT.2	T1OE	When 0, this bit allows the T1 pin to clock Timer 1 when in the counter mode. When 1, T1 acts as an output and toggles at every Timer 1 overflow.
TSTAT.1	T0RD	This bit is set, for Timer 0, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
TSTAT.0	T0OE	When 0, this bit allows the T0 pin to clock Timer 0 when in the counter mode. When 1, T0 acts as an output and toggles at every Timer 0 overflow.

SU00612A

Figure 5. Timer 0 And 1 Extended Status (TSTAT)

T2MOD Address:419
Bit Addressable
Reset Value: 00H

MSB							LSB
—	—	RCLK1	TCLK1	—	T2RD	T2OE	DCEN

BIT	SYMBOL	FUNCTION
T2MOD.5	RCLK1	Receive Clock Flag.
T2MOD.4	TCLK1	Transmit Clock Flag. RCLK1 and TCLK1 are used to select Timer 2 overflow rate as a clock source for UART1.
T2MOD.2	T2RD	This bit is set for Timer 2, if TL rolls over and clocks TH before the entire 16-bit read sequence is complete.
T2MOD.1	T2OE	When 0, this bit allows the T2 pin to clock Timer 2 when in the counter mode. When 1, T2 acts as an output and toggles at every Timer 2 overflow.
T2MOD.0	DCEN	Controls count direction for Timer 2 in autoreload mode. DCEN=1 counter set to count up DCEN=0 counter set to count down

SU00610

Figure 6. Timer 2 Mode Control (T2MOD)

WATCHDOG TIMER

The watchdog timer subsystem protects the system from incorrect code execution by causing a system reset when the watchdog timer underflows as a result of a failure of software to feed the timer prior to the timer reaching its terminal count.

Watchdog Function

The watchdog consists of a programmable prescaler and the main timer. The prescaler derives its clock from the on-chip oscillator. The prescaler consists of a programmable TCLK followed by a 13 stage counter with taps from stage 6 through stage 13. This is shown in Figure 7. The tap selection is also programmable. The watchdog main counter is a down counter clocked (decremented) each time the programmable prescaler underflows. The watchdog generates an underflow signal (and is autoloaded) when the watchdog is at count 0 and the clock to decrement the watchdog occurs. The watchdog is 8 bits long and the autoload value can range from 0 to FFH. (The autoload value of 0 is permissible since the prescaler is cleared upon autoload).

This leads to the following user design equations. Definitions :t_{OSC} is the oscillator period, N is the selected prescaler tap value, W is the main counter autoload value, t_{MIN} is the minimum watchdog time-out value (when the autoload value is 0), t_{MAX} is the maximum time-out value (when the autoload value is FFH), t_D is the design time-out value.

$t_{MIN} = t_{OSC} \times 4 \times 64$ (W = 0)

$t_{MAX} = t_{OSC} \times 64 \times 8192 \times 256$ (W = 255)

$t_D = t_{MIN} \times 2^{PRESCALER} \times (W + 1)$
(where prescaler = 0, 1, 2, 3, 4, 5, 6, or 7)

The watchdog timer is not directly loadable by the user. Instead, the value to be loaded into the main timer is held in an autoload register or is part of the mask ROM programming. In order to cause the main timer to be loaded with the appropriate value, a special sequence of software action must take place. This operation is referred to as feeding the watchdog timer.

To feed the watchdog, two instructions must be sequentially executed successfully. No intervening SFR accesses are allowed, so interrupts should be disabled before feeding the watchdog. The instructions should move A5H to the WFEED1 register and then 5AH to the WFEED2 register. If WFEED1 is correctly loaded and WFEED2 is not correctly loaded, then an immediate watchdog reset will occur.

The software must be written so that a feed operation takes place every t_D seconds from the last feed operation. Some tradeoffs may need to be made. It is not advisable to include feed operations in minor loops or in subroutines unless the feed operation is a specific subroutine.

Watchdog Control Register (WDCON) (Bit Addressable)

The following bits of this register are read only in the ROM part when $\overline{EA}$ is high: PRE0, PRE1, and PRE2. That is, the register will reflect the mask programmed values. In the ROM part with $\overline{EA}$ high, these bits are taken from mask coded bits and are not readable by the program.

The reset values of the WDCON and WDL registers will be such that the watchdog timer has a timeout period of $4 \times 64 \times t_{OSC}$. The watchdog timer will not generate an interrupt. WDCON can be written by software only by executing a valid watchdog feed sequence.

The watchdog timer subsystem consists of a programmable 13-bit prescaler, and an 8-bit main timer. The main timer is clocked by a tap taken from one of the top 8-bits of the prescaler. The clock source for the prescaler is the same as TCLK (same as the clock source for the timers). Thus the main counter can be clocked as often as once every 64* TCLKs (see Table 1).

Table 1. Prescaler Select Values in WDCON

PRE2	PRE1	PRE0	DIVISOR
0	0	0	TCLK*32*2
0	0	1	TCLK*32*4
0	1	0	TCLK*32*8
0	1	1	TCLK*32*16
1	0	0	TCLK*32*32
1	0	1	TCLK*32*64
1	1	0	TCLK*32*128
1	1	1	TCLK*32*256

NOTE:
Where, $t_{CLK} = t_{OSC} \times 4, \times 16, \times 64$ (set in SCR).

Programming the Watchdog Timer

Both the EPROM and ROM devices have a set of SFRs for holding the watchdog autoload values and the control bits. The watchdog time-out flag is present in the watchdog control register and operates the same in all versions. In the EPROM device, the watchdog parameters (autoload value and control) are always taken from the SFRs. In the ROM device, the watchdog parameters can be mask programmed or taken from the SFRs. The selection to take the watchdog parameters from the SFRs or from the mask programmed values is controlled by $\overline{EA}$ (external access). When $\overline{EA}$ is high (internal ROM access), the watchdog parameters are taken from the mask programmed values. When $\overline{EA}$ is low (external access), the watchdog parameters are taken from the SFRs. The user should be able to leave code in his program which initializes the watchdog SFRs even though he has migrated to the mask ROM part. This allows no code changes from EPROM prototyping to ROM coded production parts.

Watchdog Detailed Operation

EPROM Device (and ROMless Operation: $\overline{EA}$ = 0)
In the ROMless operation (ROM part, $\overline{EA}$ = 0) and in the EPROM device, the watchdog operates in the following manner.

When external $\overline{RESET}$ is applied, the following takes place:

- Watchdog run control bit set to ON.
- Autoload register WDL set to 00 (min. count).
- Watchdog time-out flag cleared.
- Prescaler is cleared.
- Prescaler tap set to the lowest divide.
- Autoload takes place.

Note that when coming out of a hardware reset, the software should load the autoload registers and then feed the watchdog (cause an autoload). The watchdog will now be starting at a known point.

If the watchdog is running and happens to underflow at the time the external RESET is applied, the watchdog time-out flag will be cleared.

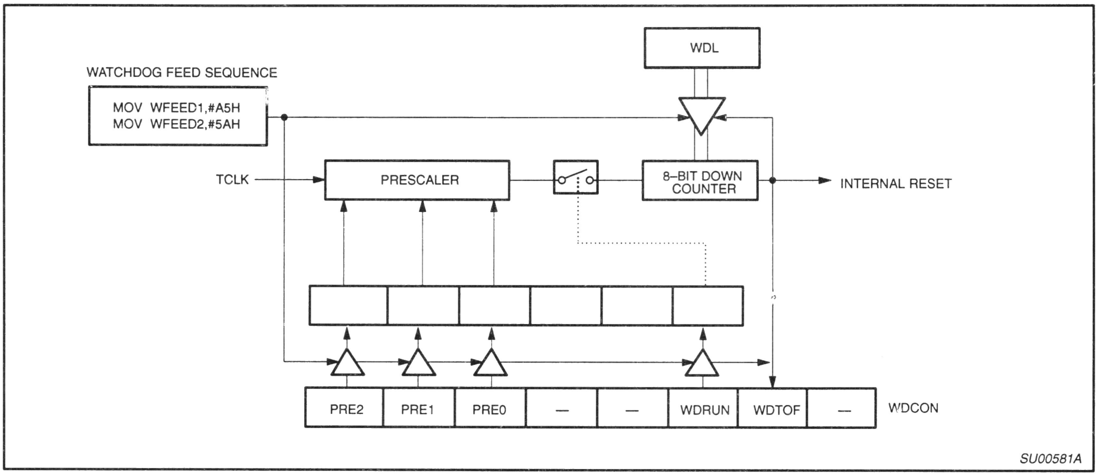


Figure 7. Watchdog Timer in XA-G3

When the watchdog underflows, the following action takes place (see Figure 7):

- Autoload takes place.
- Watchdog time-out flag is set
- Watchdog run bit unchanged.
- Autoload register unchanged.
- Prescaler tap unchanged.
- All other device action same as external reset.

Note that if the watchdog underflows, the program counter will be loaded from the reset vector as in the case of an internal reset. The watchdog time-out flag can be examined to determine if the watchdog has caused the reset condition. The watchdog time-out flag bit can be cleared by software.

WDCON Register Bit Definitions

WDCON.7	PRE2	Prescaler Select 2, reset to 1
WDCON.6	PRE1	Prescaler Select 1, reset to 1
WDCON.5	PRE0	Prescaler Select 0, reset to 1
WDCON.4	—	
WDCON.3	—	
WDCON.2	WDRUN, reset to 1	
WDCON.1	WDTOF	Timeout flag
WDCON.0	—	

UARTs

The XA-G3 includes 2 UART ports that are compatible with the enhanced UART used on the 8xC51FB. Baud rate selection is somewhat different due to the clocking scheme used for the XA timers.

Some other enhancements have been made to UART operation. The first is that there are separate interrupt vectors for each UART's transmit and receive functions. The second is double-buffering of the transmit register to allow time for interrupt processing without introducing inter-character gaps when tightly transmitted characters are required in the application. A break detect function has been added to the UART. This operates independently of the UART itself and provides a start-of-break status bit that the program may test. Finally, an Overrun Error flag has been added to detect missed characters in the received data stream.

Each UART rate is determined by either a fixed division of the oscillator (in UART modes 0 and 2) or by the timer 1 or timer 2 overflow rate (in UART modes 1 and 3).

The serial port receive and transmit registers are both accessed at Special Function Register SnBUF. Writing to SnBUF loads the transmit register, and reading SnBUF accesses a physically separate receive register.

The serial port can operate in 4 modes:

Mode 0: Serial I/O expansion mode. Serial data enters and exits through RxDn. TxDn outputs the shift clock. 8 bits are transmitted/received (LSB first). (The baud rate is fixed at 1/16 the oscillator frequency.)

Mode 1: Standard 8-bit UART mode. 10 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), and a stop bit (1). On receive, the stop bit goes into RB8 in Special Function Register SnCON. The baud rate is variable.

Mode 2: Fixed rate 9-bit UART mode. 11 bits are transmitted (through TxD) or received (through RxD): start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). On Transmit, the 9th data bit (TB8_n in SnCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8_n. On receive, the 9th data bit goes into RB8_n in Special Function Register SnCON, while the stop bit is ignored. The baud rate is programmable to 1/32 of the oscillator frequency.

Mode 3: Standard 9-bit UART mode. 11 bits are transmitted (through TxDn) or received (through RxDn): a start bit (0), 8 data bits (LSB first), a programmable 9th data bit, and a stop bit (1). In fact, Mode 3 is the same as Mode 2 in all respects except baud rate. The baud rate in Mode 3 is variable.

In all four modes, transmission is initiated by any instruction that uses SnBUF as a destination register. Reception is initiated in Mode 0 by the condition RI_n = 0 and REN_n = 1. Reception is initiated in the other modes by the incoming start bit if REN_n = 1.

Serial Port Control Register

The serial port control and status register is the Special Function Register SnCON, shown in Figure 9. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8_n and RB8_n), and the serial port interrupt bits (TI_n and RI_n).

CLOCKING SCHEME/BAUD RATE GENERATION

The XA UARTS clock rates are determined by either a fixed division (modes 0 and 2) of the oscillator clock or by the Timer 1 or Timer 2 overflow rate (modes 1 and 3).

The clock for the UARTs in XA runs at 16x the Baud rate. If the timers are used as the source for Baud Clock, since maximum speed of timers/Baud Clock is Osc/4, the maximum baud rate is timer overflow divided by 16 i.e. Osc/64.

In Mode 0, it is fixed at Osc/16. In Mode 2, however, the fixed rate is Osc/32.

Pre-scaler for all Timers T0,1,2 controlled by PT1, PT0 bits in SCR	00	Osc/4
	01	Osc/16
	10	Osc/64
	11	reserved

Baud Rate for UART Mode 0:

Baud_Rate = Osc/16

Baud Rate calculation for UART Mode 1 and 3:

Baud_Rate = Timer_Rate/16*N

Timer_Rate = Osc/(N*(Timer_Range– Timer_Reload_Value))

where N = the TCLK prescaler value: 4, 16, or 64.
and Timer_Range = 256 for timer 1 in mode 2.
65536 for timer 1 in mode 0 and timer 2 in count up mode.

The timer reload value may be calculated as follows:

Timer_Reload_Value = Timer_Range–(Osc/(Baud_Rate*N*16))

NOTES:

1. The maximum baud rate for a UART in mode 1 or 3 is Osc/64.
2. The lowest possible baud rate (for a given oscillator frequency and N value) may be found by using a timer reload value of 0.
3. The timer reload value may never be larger than the timer range.
4. If a timer reload value calculation gives a negative or fractional result, the baud rate requested is not possible at the given oscillator frequency and N value.

Baud Rate for UART Mode 2:

Baud_Rate = Osc/32

Using Timer 2 to Generate Baud Rates

Timer T2 is a 16-bit up/down counter in XA. As a baud rate generator, timer 2 is selected as a clock source for either/both UART0 and UART1 transmitters and/or receivers by setting TCLKn and/or RCLKn in T2CON and T2MOD. As the baud rate generator, T2 is incremented as Osc/N where N = 4, 16 or 64 depending on TCLK as programmed in the SCR bits PT1, and PTO. So, if T2 is the source of one UART, the other UART could be clocked by either T1 overflow or fixed clock, and the UARTs could run independently with different baud rates.

T2CON 0x418		bit5	bit4	
		RCLK0	TCLK0	

T2MOD 0x419		bit5	bit4	
		RCLK1	TCLK1	

When Timer 1 or 2 is the source for baud clock, the baud rate is given by

Baud Rate = Osc/16*1/Timer Overflow Rate

The timer T2 or T1 (16-bit mode) reload value is set by

Up-counter Mode

reload = 65,536–[Osc/N)*1/Baud Rate]
where N = 4,16 or 64

Down-counter Mode

reload = [(Osc/16N)*1/Baud Rate]
where N = 4, 16, or 64

Prescaler Select for Timer Clock (TCLK)

SCR 0x440		bit3	bit2	
		PT1	PT0	

SnSTAT Address: S0STAT 421
S1STAT 425
Bit Addressable
Reset Value: 00H

MSB LSB

—	—	—	—	FEn	BRn	OEn	STINTn

BIT	SYMBOL	FUNCTION
SnSTAT.3	FEn	Framing Error flag is set when the receiver fails to see a valid STOP bit at the end of the frame.
SnSTAT.2	BRn	Break Detect flag is set if a character is received with all bits (including STOP bit) being logic '0'. Thus it gives a "Start of Break Detect" on bit 8 for Mode 1 and bit 9 for Modes 2 and 3. The break detect feature operates independently of the UARTs and provides the START of Break Detect status bit that a user program may poll.
SnSTAT.1	OEn	Overrun Error flag is set if a new character is received in the receiver buffer while it is still full (before the software has read the previous character from the buffer), i.e., when bit 8 of a new byte is received while RI in SnCON is still set.
SnSTAT.0	STINTn	This flag must be set to enable any of the above status flags to generate a receive interrupt (RIn). The only way it can be cleared is by a software write to this register.

SU00607A

Figure 8. Serial Port Extended Status (SnSTAT) Register
(See also Figure 10 regarding Framing Error flag)

INTERRUPT SCHEME

There are separate interrupt vectors for each UART's transmit and receive functions.

Table 2. Vector Locations for UARTs in XA

Vector Address	Interrupt Source	Arbitration
9CH – 9FH	Uart 1 Receiver	8
A0H – A3H	Uart 1 Transmitter	9
A4H – A7H	Uart 2 Receiver	10
A8H – ABH	Uart 2 Transmitter	11

NOTE:
The transmit and receive vectors could contain the same ISR address to work like a 8051 interrupt scheme

Error Handling, Status Flags and Break Detect

The UARTs in XA has the following error flags; see Figure 8.

Multiprocessor Communications

Modes 2 and 3 have a special provision for multiprocessor communications. In these modes, 9 data bits are received. The 9th one goes into RB8. Then comes a stop bit. The port can be programmed such that when the stop bit is received, the serial port interrupt will be activated only if RB8 = 1. This feature is enabled by setting bit SM2 in SCON. A way to use this feature in multiprocessor systems is as follows:

When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte differs from a data byte in that the 9th bit is 1 in an address byte and 0 in a data byte. With SM2 = 1, no slave will be interrupted by a data byte. An address byte, however, will interrupt all slaves, so that each slave can examine the received byte and see if it is being addressed. The addressed slave will clear its SM2 bit and prepare to receive the data bytes that will be coming. The slaves that weren't being addressed leave their SM2s set and go on about their business, ignoring the coming data bytes.

SM2 has no effect in Mode 0, and in Mode 1 can be used to check the validity of the stop bit although this is better done with the Framing Error (FE) flag. In a Mode 1 reception, if SM2 = 1, the receive interrupt will not be activated unless a valid stop bit is received.

Automatic Address Recognition

Automatic Address Recognition is a feature which allows the UART to recognize certain addresses in the serial bit stream by using hardware to make the comparisons. This feature saves a great deal of software overhead by eliminating the need for the software to examine every serial address which passes by the serial port. This feature is enabled by setting the SM2 bit in SCON. In the 9 bit UART modes, mode 2 and mode 3, the Receive Interrupt flag (RI) will be automatically set when the received byte contains either the "Given" address or the "Broadcast" address. The 9 bit mode requires that the 9th information bit is a 1 to indicate that the received information is an address and not data. Automatic address recognition is shown in Figure 11.

Using the Automatic Address Recognition feature allows a master to selectively communicate with one or more slaves by invoking the Given slave address or addresses. All of the slaves may be contacted by using the Broadcast address. Two special Function Registers are used to define the slave's address, SADDR, and the address mask, SADEN. SADEN is used to define which bits in the SADDR are to be used and which bits are "don't care". The SADEN mask can be logically ANDed with the SADDR to create the "Given" address which the master will use for addressing each of the slaves. Use of the Given address allows multiple slaves to be recognized while excluding others. The following examples will help to show the versatility of this scheme:

Slave 0 SADDR = 1100 0000
SADEN = 1111 1101
Given = 1100 00X0

Slave 1 SADDR = 1100 0000
SADEN = 1111 1110
Given = 1100 000X

In the above example SADDR is the same and the SADEN data is used to differentiate between the two slaves. Slave 0 requires a 0 in bit 0 and it ignores bit 1. Slave 1 requires a 0 in bit 1 and bit 0 is ignored. A unique address for Slave 0 would be 1100 0010 since slave 1 requires a 0 in bit 1. A unique address for slave 1 would be 1100 0001 since a 1 in bit 0 will exclude slave 0. Both slaves can be selected at the same time by an address which has bit 0 = 0 (for slave 0) and bit 1 = 0 (for slave 1). Thus, both could be addressed with 1100 0000.

In a more complex system the following could be used to select slaves 1 and 2 while excluding slave 0:

Slave 0 SADDR = 1100 0000
SADEN = 1111 1001
Given = 1100 0XX0

Slave 1 SADDR = 1110 0000
SADEN = 1111 1010
Given = 1110 0X0X

Slave 2 SADDR = 1110 0000
SADEN = 1111 1100
Given = 1110 00XX

In the above example the differentiation among the 3 slaves is in the lower 3 address bits. Slave 0 requires that bit 0 = 0 and it can be uniquely addressed by 1110 0110. Slave 1 requires that bit 1 = 0 and it can be uniquely addressed by 1110 and 0101. Slave 2 requires that bit 2 = 0 and its unique address is 1110 0011. To select Slaves 0 and 1 and exclude Slave 2 use address 1110 0100, since it is necessary to make bit 2 = 1 to exclude slave 2.

The Broadcast Address for each slave is created by taking the logical OR of SADDR and SADEN. Zeros in this result are teated as don't-cares. In most cases, interpreting the don't-cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR and SADEN are loaded with 0s. This produces a given address of all "don't cares" as well as a Broadcast address of all "don't cares". This effectively disables the Automatic Addressing mode and allows the microcontroller to use standard UART drivers which do not make use of this feature.

SnCON Address: S0CON 420
S1CON 424

Bit Addressable
Reset Value: 00H

MSB							LSB
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Where SM0, SM1 specify the serial port mode, as follows:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	shift register	fosc/ 16
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	fosc/32
1	1	3	9-bit UART	variable

BIT	SYMBOL	FUNCTION
SnCON.5	SM2	Enables the multiprocessor communication feature in Modes 2 and 3. In Mode 2 or 3, if SM2 is set to 1, then RI will not be activated if the received 9th data bit (RB8) is 0. In Mode 1, if SM2=1 then RI will not be activated if a valid stop bit was not received. In Mode 0, SM2 should be 0.
SnCON.4	REN	Enables serial reception. Set by software to enable reception. Clear by software to disable reception.
SnCON.3	TB8	The 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired.
SnCON.2	RB8	In Modes 2 and 3, is the 9th data bit that was received. In Mode 1, it SM2=0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
SnCON.1	TI	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
SnCON.0	RI	Receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

SU00597A

Figure 9. Serial Port Control (SnCON) Register

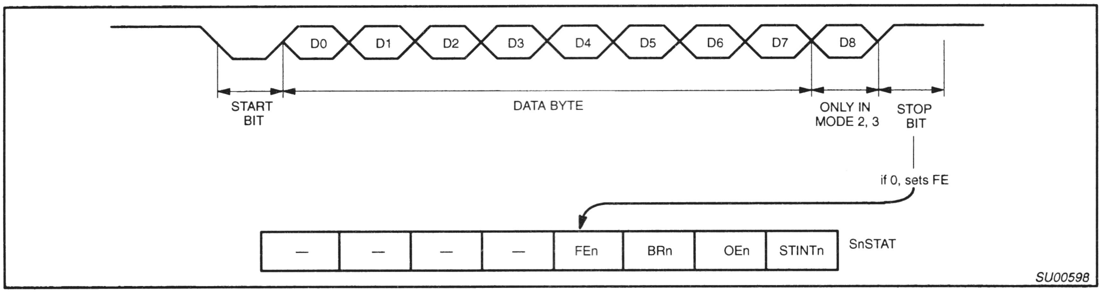
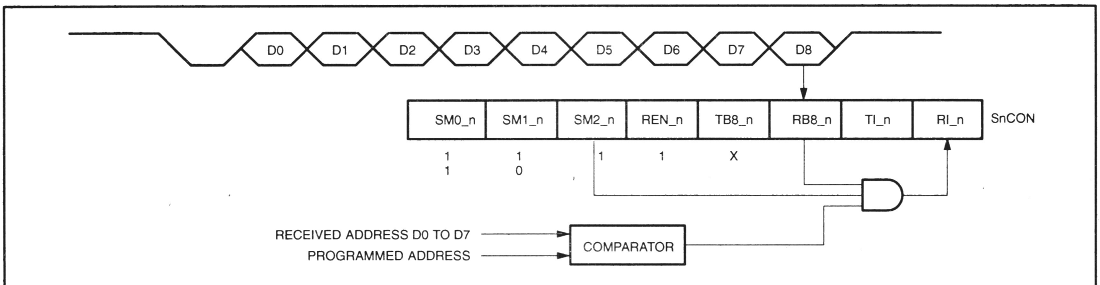

D0
D1
D2
D3
D4
D5
D6
D7
D8
START BIT
DATA BYTE
ONLY IN MODE 2, 3
STOP BIT
if 0, sets FE
—
—
—
—
FEn
BRn
OEn
STINTn
SnSTAT
SU00598

Figure 10. UART Framing Error Detection


D0
D1
D2
D3
D4
D5
D6
D7
D8
SM0_n
SM1_n
SM2_n
REN_n
TB8_n
RB8_n
TI_n
RI_n
SnCON
1
1
1
1
X
1
0
RECEIVED ADDRESS D0 TO D7
PROGRAMMED ADDRESS
COMPARATOR

IN UART MODE 2 OR MODE 3 AND SM2 = 1:
INTERRUPT IF REN=1, RB8=1 AND "RECEIVED ADDRESS" = "PROGRAMMED ADDRESS"
– WHEN OWN ADDRESS RECEIVED, CLEAR SM2 TO RECEIVE DATA BYTES
– WHEN ALL DATA BYTES HAVE BEEN RECEIVED: SET SM2 TO WAIT FOR NEXT ADDRESS.

SU00613

Figure 11. UART Multiprocessor Communication, Automatic Address Recognition

I/O PORT OUTPUT CONFIGURATION

Each I/O port pin on the XA-G3 can be user configured to one of 4 output types. The types are Quasi-bidirectional (essentially the same as standard 80C51 family I/O ports), Open-Drain, Push-Pull, and Off (high impedance). The default configuration after reset is Quasi-bidirectional. However, in the ROMless mode (the $\overline{EA}$ pin is low at reset), the port pins that comprise the external data bus will default to push-pull outputs.

I/O port output configurations are determined by the settings in port configuration SFRs. There are 2 SFRs for each port, called PnCFGA and PnCFGB, where "n" is the port number. One bit in each of the 2 SFRs relates to the output setting for the corresponding port pin, allowing any combination of the 2 output types to be mixed on those port pins. For instance, the output type of port 1 pin 3 is controlled by the setting of bit 3 in the SFRs P1CFGA and P1CFGB.

Table 3 shows the configuration register settings for the 4 port output types. The electrical characteristics of each output type may be found in the DC Characteristic table.

Table 3. Port Configuration Register Settings

PnCFGB	PnCFGA	Port Output Mode
0	0	Open Drain
0	1	Quasi-bidirectional
1	0	Off (high impedance)
1	1	Push-Pull

NOTE:
Mode changes may cause glitches to occur during transitions. When modifying both registers, WRITE instructions should be carried out consecutively.

EXTERNAL BUS

The external program/data bus on the XA-G3 allows for 8-bit or 16-bit bus width, and address sizes from 12 to 20 bits. The bus width is selected by an input at reset (see Reset Options below), while the address size is set by the program in a configuration register. If all off-chip code is selected (through the use of the $\overline{EA}$ pin), the initial code fetches will be done with the maximum address size (20 bits).

RESET

The device is reset whenever a logic "0" is applied to $\overline{RST}$ for at least 10 microseconds, placing a low level on the pin re-initializes the on-chip logic.

The duration of reset must be extended when power is initially applied or when using reset to exit power down mode. This is due to the need to allow the oscillator time to start up and stabilize. For most power supply ramp up conditions, this time is 10 milliseconds.

As it is brought high again, an exception is generated which causes the processor to jump to the address contained in the memory location 0000. The destination of the reset jump must be located in the first 64k of code address on power-up, all vectors are 16-bit values and so point to page zero addresses only. After a reset the RAM contents are indeterminate.

RESET OPTIONS

The $\overline{EA}$ pin is sampled on the rising edge of the $\overline{RST}$ pulse, and determines whether the device is to begin execution from internal or external code memory. $\overline{EA}$ pulled high configures the XA in single-chip mode. If $\overline{EA}$ is driven low, the device enters ROMless mode. After Reset is released, the $\overline{EA}$/WAIT pin becomes a bus wait signal for external bus transactions.

The BUSW/P3.5 pin is weakly pulled high while reset is asserted, allowing simple biasing of the pin with a resistor to ground to select the alternate bus width.

POWER REDUCTION MODES

The XA-G3 supports Idle and Power down modes of power reduction. The idle mode leaves some peripherals running to allow them to wake up the processor when an interrupt is generated. The power down mode stops the processor clock in order to absolutely minimize power. The processor can be made to exit power down mode via reset or one of the external interrupt inputs. In power down mode, the power supply voltage may be further reduced to the keep-alive voltage, retaining the RAM, register, and SFR values at the point where the power down mode was entered.

INTERRUPTS

The XA-G3 supports 31 maskable interrupts vectored interrupt sources. The maskable interrupts each have 16 priority levels and may be globally and/or individually enabled or disabled.

The XA defines four types of interrupts:

- **Exception Interrupts** – These are system level errors and other very important occurrences which include stack overflow, divide-by-0, and reset.
- **Event interrupts** – These are peripheral interrupts from devices such as UARTs, timers, and external interrupt inputs.
- **Software Interrupts** – These are equivalent of hardware interrupt, but are requested only under software control.
- **Trap Interrupts** – These are TRAP instructions, generally used to call system services in a multi-tasking system.

Exception interrupts, software interrupts, and trap interrupts are generally standard for XA derivatives and are detailed in the XA User Guide. Event interrupts tend to be different on different XA derivatives.

The XA-G3 supports a total of 9 maskable event interrupt sources (for the various XA-G3 peripherals), seven software interrupts, 5 exception interrupts (plus reset), and 16 traps. The maskable event interrupts share a global interrupt enable bit (the EA bit in the IEL register) and each also has a separate individual interrupt enable bit (in the IEL or IEH registers). Each event interrupt can be set to occur at one of 8 priority levels (levels 8 through 15) via bits in the Interrupt Priority (IP) registers, IPA0 through IPA5. Details of the priority scheme may be found in the XA User Guide.

The complete interrupt vector list for the XA-G3, including all 4 interrupt types, is shown in the following tables. The tables include the address of the vector for each interrupt, the related priority register bits (if any), and the arbitration ranking for that interrupt source. The arbitration ranking determines the order in which interrupts are processed if more than one interrupt of the same priority occurs simultaneously.

Table 4. Interrupt Vectors

EXCEPTION/TRAPS PRECEDENCE

DESCRIPTION	VECTOR ADDRESS	ARBITRATION RANKING
Reset (h/w, watchdog, s/w)	0000–0003	0 (High)
Breakpoint (h/w trap 1)	0004–0007	1
Trace (h/w trap 2)	0008–000B	1
Stack Overflow (h/w trap 3)	000C–000F	1
Divide by 0 (h/w trap 4)	0010–0013	1
User RETI (h/w trap 5)	0014–0017	1
TRAP 0– 15 (software)	0040–007F	1

EVENT INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY	ARBITRATION RANKING
External interrupt 0	IE0	0080–0083	EX0	IPA0.3–0	2
Timer 0 interrupt	TF0	0084–0087	ET0	IPA0.7–4	3
External interrupt 1	IE1	0088–008B	EX1	IPA1.3–0	4
Timer 1 interrupt	TF1	008C–008F	ET1	IPA1.7–4	5
Timer 2 interrupt	TF2	0090–0093	ET2	IPA2.3–0	6
Serial port 0 Rx	RI.0	00A0–00A3	ERI0	IPA4.3–0	7
Serial port 0 Tx	TI.0	00A4–00A7	ETI0	IPA4.7–4	8
Serial port 1 Rx	RI.1	00A8–00AB	ERI1	IPA5.3–0	9
Serial port 1 Tx	TI.1	00AC–00AF	ETI1	IPA5.7–4	10

SOFTWARE INTERRUPTS

DESCRIPTION	FLAG BIT	VECTOR ADDRESS	ENABLE BIT	INTERRUPT PRIORITY
Software interrupt 1	SWR1	0100–0103	SWE1	(fixed at 1)
Software interrupt 2	SWR2	0104–0107	SWE2	(fixed at 2)
Software interrupt 3	SWR3	0108–010B	SWE3	(fixed at 3)
Software interrupt 4	SWR4	010C–010F	SWE4	(fixed at 4)
Software interrupt 5	SWR5	0110–0113	SWE5	(fixed at 5)
Software interrupt 6	SWR6	0114–0117	SWE6	(fixed at 6)
Software interrupt 7	SWR7	0018–011B	SWE7	(fixed at 7)

ABSOLUTE MAXIMUM RATINGS

PARAMETER	RATING	UNIT
Operating temperature under bias	–55 to +125	°C
Storage temperature range	–65 to +150	°C
Voltage on $\overline{EA}/V_{PP}$ pin to V_{SS}	0 to +13.0	V
Voltage on any other pin to V_{SS}	–0.5 to V_{DD}+0.5V	V
Maximum I_{OL} per I/O pin	15	mA
Power dissipation (based on package heat transfer limitations, not device power consumption)	1.5	W

DC ELECTRICAL CHARACTERISTICS

V_{DD} = 5.0V ±10% to 3.0V ±10% unless otherwise specified;
T_{amb} = T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial, unless otherwise specified.

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS			UNIT
			MIN	TYP	MAX	
Supplies						
I_{DD}	Supply current operating	5.0V, 30 MHz			100	mA
I_{ID}	Idle mode supply current	5.0V, 30 MHz			25	mA
I_{PD}	Power-down current	5.0V, 3.0V		5	50	µA
V_{RAM}	RAM-keep-alive voltage	Ram-keep-alive voltage	1.5			V
V_{IL}	Input low voltage, except		–0.5		0.8	V
V_{IH}	Input high voltage, except XTAL1, RST	At 5.0V[1]	2.2			V
		At 3.0V[1]	2			V
V_{IH1}	Input high voltage to XTAL1, RST	For both 3.0 & 5.0V	0.7V_{DD}			V
V_{OL}	Output low voltage all ports, ALE, PSEN[5]	I_{OL} = 3.2mA, V_{DD} = 5.0V			0.8	V
		1.0mA, V_{DD} = 3.0V				V
V_{OH1}	Output high voltage all ports, ALE, PSEN[3]	I_{OH} = –100µA, V_{DD} = 5.0V	2.4			V
		I_{OH} = –30µA, V_{DD} = 3.0V	2.2			V
V_{OH2}	Output high voltage, ports P0–3, ALE, PSEN[4]	I_{OH} = 3.2mA, V_{DD} = 5.0V	2.4			V
		I_{OH} = 1mA, V_{DD} = 3.0V	2.2			V
C_{IO}	Input/Output pin capacitance[2]				15	pF
I_{IL}	Logical 0 input current, P0–3[8]	V_{IN} = 0.45V			–50	µA
I_{LI}	Input leakage current, P0–3[7]				±10	µA
I_{TL}	Logical 1 to 0 transition current all ports[6]	At 6V			–650	µA
		At 3V			–250	µA

NOTE:

1. Values are linear in between
2. Max. 15 pF for~EA/V_{PP}
3. Ports in Quasi bi-directional mode with weak pull-up
4. Ports in Push-Pull mode, both pull-up and pull-down assumed to be same strength
5. In all output modes
6. Port pins source a transition current when used in quasi-bidirectional mode and externally driven from 1 to 0. This current is highest when V_{IN} is approximately 2V.
7. Measured with port in high impedance output mode.
8. Measured with port in quasi-bidirectional output mode.
9. Load capacitance for port 0, ALE, and $\overline{PSEN}$=100pF, load capacitance for all other outputs=80pF.
10. Under steady state (non-transient) conditions, I_{OL} must be extremely limited as follows:

Maximum I_{OL} per port pin:	15mA (*NOTE: This is 85°C specification.)
Maximum I_{OL} per 8-bit port:	26mA
Maximum total I_{OL} for all output:	71mA

If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.

AC ELECTRICAL CHARACTERISTICS

V_{DD} = 5.0V ±10% or 3.0V ±10%, T_{amb} = 0 to +70°C for commercial, –40°C to +85°C for industrial.

SYMBOL	PARAMETER	30MHz[1]			16MHz[2]			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
External Clock								
f_C	Oscillator frequency		30			16		MHz
$t_C = 1/f_C$	Clock period and CPU timing cycle		$1/f_C$			$1/f_C$		ns
t_{CHCX}	Clock high-time (60%–40% duty cycle)		$(t_C/2) * 0.4$			$(t_C/2) * 0.4$		ns
t_{CLCX}	Clock low-time (60%–40% duty cycle)		$(t_C/2) * 0.4$			$(t_C/2) * 0.4$		ns
t_{CLCH}	Clock rise-time		5			10		ns
t_{CHCL}	Clock fall-time		5			10		ns
Address Cycle								
t_{CRAR}	Delay from clock rising edge to ALE rising edge		24			38		ns
t_{LHLL}	ALE pulse width (programmable)		$(N+0.5) * t_C$			$(N+0.5) * t_C$		ns
t_{AVLL}	Address valid to ALE de-asserted (set-up)		t_{LLHL}			$t_{LLHL} - 4$		ns
t_{LLAX}	Address hold after ALE de-asserted		12			30		ns
Code Read Cycle								
t_{LLPL}	ALE de-asserted to PSEN active		$(t_C/2) + 10$			$(t_C/2) + 1$		ns
t_{AVIV}	Address valid to instruction valid (access time)		$(M*t_C) - 16$			$(M*t_C) - 21$		ns
t_{PLIV}	$\overline{PSEN}$ low to instruction valid		$(O*t_C) - 16$			$(O*t_C) - 21$		ns
t_{PLPH}	$\overline{PSEN}$ pulse width		$(O*t_C) - 5$			$O*t_C$		ns
t_{PXIX}	Instruction hold after $\overline{PSEN}$ de-asserted	0			0			ns
t_{PXIZ}	Bus 3-State after $\overline{PSEN}$ de-asserted		29			54		ns
t_{UAPH}	Hold time of unlatched port of address after $\overline{PSEN}$ is de-asserted.	0			0			ns
Data Read Cycle								
t_{RLRH}	$\overline{RD}$ pulse width		$(P*t_C) - 8$			$P*t_C$		ns
t_{LLRL}	ALE falling edge to $\overline{RD}$ falling edge		$(t_C/2) + 9$			$(t_C/2) + 1$		ns
t_{AVDV}	Data input valid after address valid (access time)		$(P*t_C) - 16$			$(P*t_C) - 21$		ns
t_{RLDV}	$\overline{RD}$ low to valid data in, enable time		$(P*t_C) - 16$			$(P*t_C) - 21$		ns
t_{RHDX}	Data hold time after $\overline{RD}$ de-asserted	0			0			ns
t_{RHDZ}	Bus 3-State after $\overline{RD}$ de-asserted		29			54		ns
t_{UARH}	Hold time of unlatched port of address after $\overline{RD}$ is de-asserted.	0			0			ns
Data Write Cycle								
t_{WLWH}	$\overline{WR}$ pulse width		$Q*t_C$			$Q*t_C$		ns
t_{LLWL}	ALE falling edge to $\overline{WR}$ asserted		$(t_C/2) + 5$			$(t_C/2) - 5$		ns
t_{QVWX}	Data valid before $\overline{WR}$ active (setup time)		$(R*t_C) - 7$			$(R*t_C) - 12$		ns
t_{WHQX}	Data hold time after $\overline{WR}$ rising edge		0			4		ns
t_{AVWL}	address valid to $\overline{WR}$ active		$(R*t_C) - 3$			$(R*t_C) - 8$		ns
t_{UAWH}	Hold time of unlatched part of address after $\overline{WR}$ is de-asserted		0			3		ns
WAIT Input								
t_{WTV}	Rising edge of Wait after $\overline{RD}$, $\overline{WR}$, and $\overline{PSEN}$ falling edge		$(S*t_C) - 15$			$(S*t_C) - 14$		ns
t_{WAIT}	CPU wait state period		t_C			t_C		ns

SYMBOL	PARAMETER	30MHz[1]			16MHz[2]			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
Input Pulse Width								
t_{PWI}	Pulse width for interrupt inputs		2^*t_C			2^*t_C		ns
t_{PWT}	Pulse width for timer inputs		2^*t_C			2^*t_C		ns
Shift Register								
t_{XLXL}	Serial port clock cycle time		16^*t_C			16^*t_C		ns
t_{QVXH}	Output data setup to clock rising edge		$(2^*t_C) - 30$			$(2^*t_C) - 40$		ns
t_{XHQX}	Output data hold to clock rising edge		$(2^*t_C) - 30$			$(2^*t_C) - 40$		ns
t_{XHDX}	Input data hold after clock rising edge		0			0		ns
t_{XHDV}	Clock rising edge to input data valid		30			40		ns

NOTES:
1. All values indicated for V_{DD} = 5V ±10%. Typical values are for 20°C.
2. All values indicated for V_{DD} = 3V ±10%. Typical values are for 20°C.
3. N = ALEW bit value
   M = burst mode code read clocks
   O = PSEN clocks
   P = RD clocks
   Q = WR clocks
   R = WR setup clocks

EXPLANATION OF THE AC SYMBOLS

Each timing symbol has five characters. The first character is always 't' (= time). The other characters, depending on their positions, indicate the name of a signal or the logical status of that signal. The designations are:

A – Address
C – Clock
D – Input data
H – Logic level high
I – Instruction (program memory contents)
L – Logic level low, or ALE
P – $\overline{\text{PSEN}}$
Q – Output data
R – $\overline{\text{RD}}$ signal
t – Time
U – Undefined
V – Valid
W– $\overline{\text{WR}}$ signal
X – No longer a valid logic level
Z – Float

Examples: t_{AVLL} = Time for address valid to ALE low.
t_{LLPL} =Time for ALE low to $\overline{\text{PSEN}}$ low.

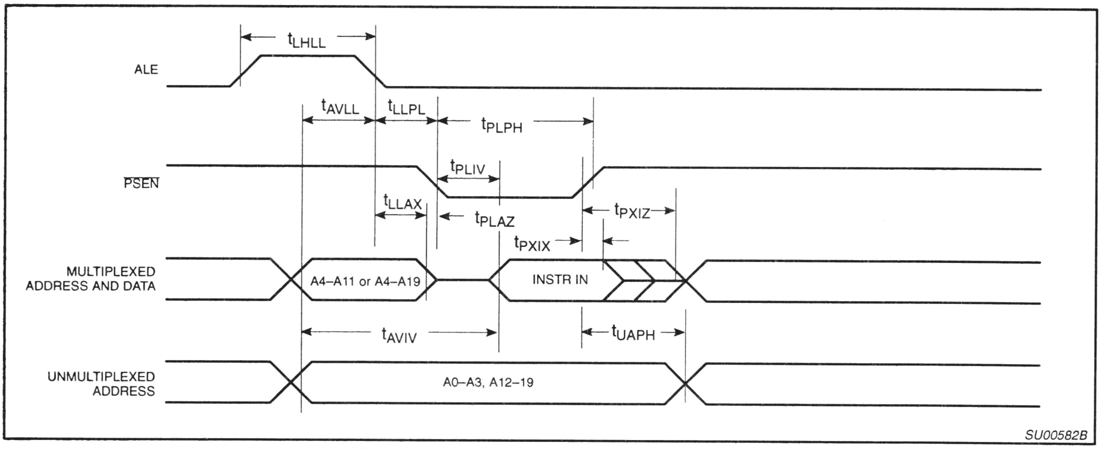

t_{LHLL}
ALE
t_{AVLL}
t_{LLPL}
t_{PLPH}
t_{PLIV}
$\overline{\text{PSEN}}$
t_{LLAX}
t_{PLAZ}
t_{PXIZ}
t_{PXIX}
MULTIPLEXED ADDRESS AND DATA
A4–A11 or A4–A19
INSTR IN
t_{AVIV}
t_{UAPH}
UNMULTIPLEXED ADDRESS
A0–A3, A12–19
SU00582B

Figure 12. External Program Memory Read Cycle

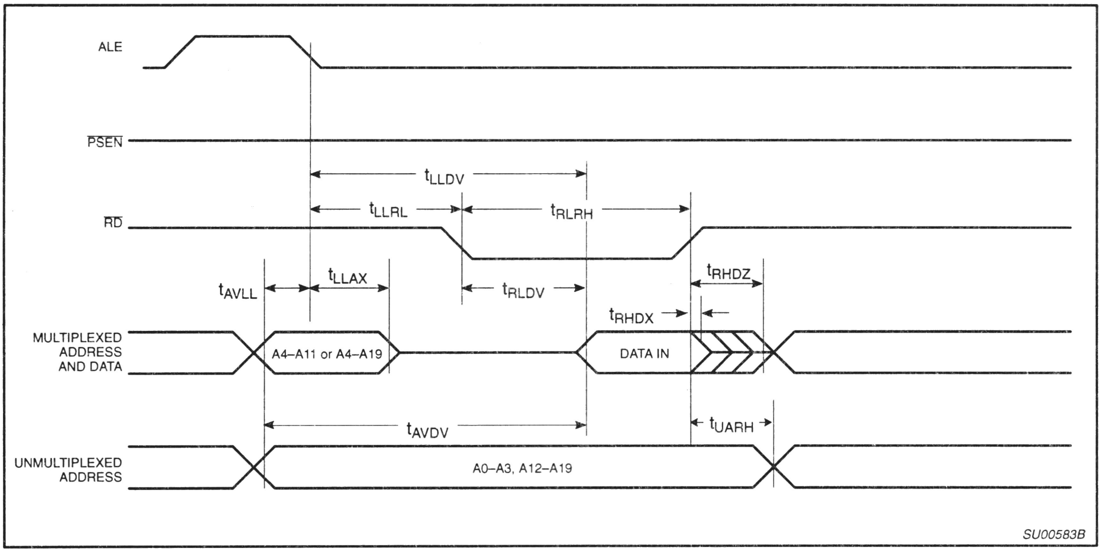

ALE
PSEN
t_{LLDV}
t_{LLRL}
t_{RLRH}
RD
t_{AVLL}
t_{LLAX}
t_{RLDV}
t_{RHDZ}
t_{RHDX}
MULTIPLEXED ADDRESS AND DATA
A4–A11 or A4–A19
DATA IN
t_{AVDV}
t_{UARH}
UNMULTIPLEXED ADDRESS
A0–A3, A12–A19
SU00583B

Figure 13. External Data Memory Read Cycle

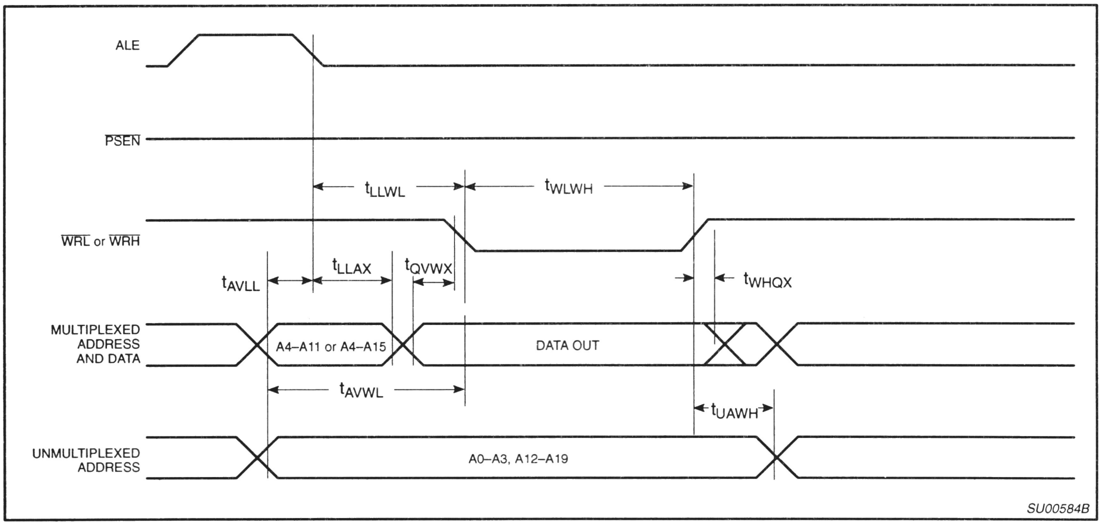

ALE
PSEN
t_{LLWL}
t_{WLWH}
WRL or WRH
t_{AVLL}
t_{LLAX}
t_{QVWX}
t_{WHQX}
MULTIPLEXED ADDRESS AND DATA
A4–A11 or A4–A15
DATA OUT
t_{AVWL}
t_{UAWH}
UNMULTIPLEXED ADDRESS
A0–A3, A12–A19
SU00584B

Figure 14. External Data Memory Write Cycle

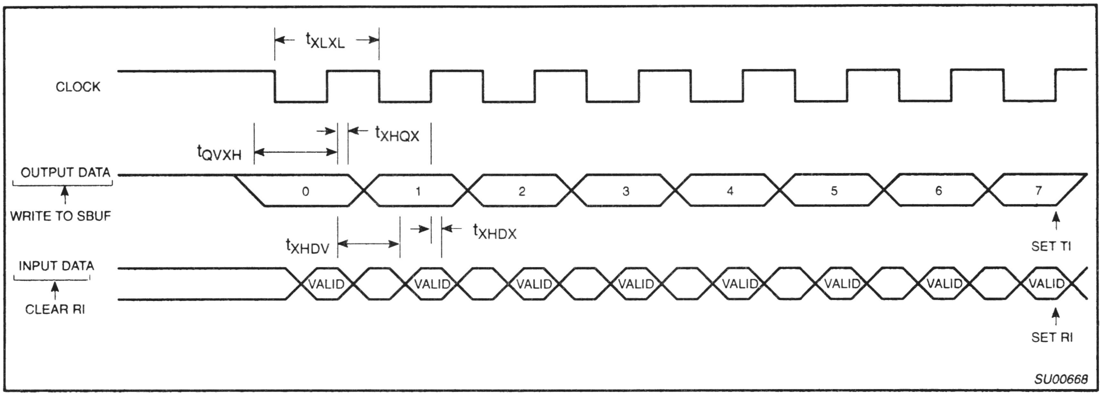

t_{XLXL}
CLOCK
t_{QVXH}
t_{XHQX}
OUTPUT DATA
WRITE TO SBUF
0
1
2
3
4
5
6
7
SET TI
t_{XHDV}
t_{XHDX}
INPUT DATA
CLEAR RI
VALID
SET RI
SU00668

Figure 15. Shift Register Mode Timing

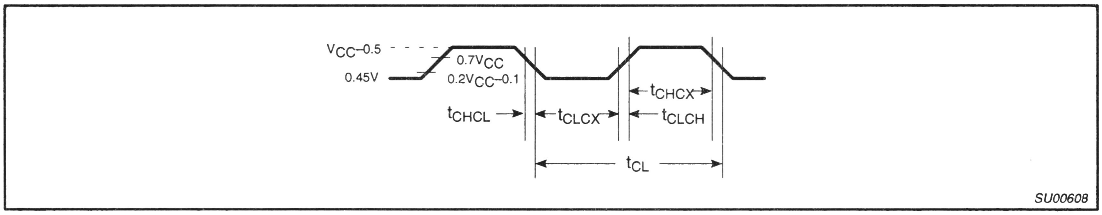
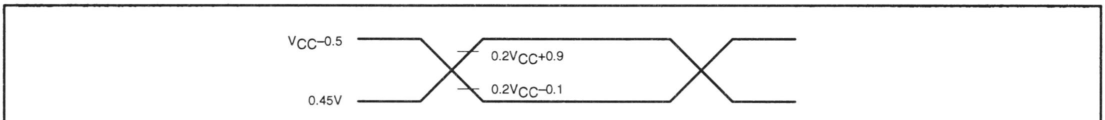

V_{CC}–0.5
0.45V
0.7V_{CC}
0.2V_{CC}–0.1
t_{CHCL}
t_{CLCX}
t_{CHCX}
t_{CLCH}
t_{CL}
SU00608

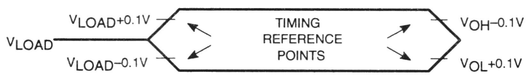
Figure 16. External Clock Drive


V_{CC}–0.5
0.45V
0.2V_{CC}+0.9
0.2V_{CC}–0.1

NOTE:
AC inputs during testing are driven at V_{CC} –0.5 for a logic '1' and 0.45V for a logic '0'.
Timing measurements are made at V_{IH} min for a logic '1' and V_{IL} max for a logic '0'.

SU00010

Figure 17. AC Testing Input/Output


V_{LOAD}+0.1V
V_{LOAD}
V_{LOAD}–0.1V
TIMING
REFERENCE
POINTS
V_{OH}–0.1V
V_{OL}+0.1V

NOTE:
For timing purposes, a port is no longer floating when a 100mV change from load voltage occurs, and begins to float when a 100mV change from the loaded V_{OH}/V_{OL} level occurs. $I_{OH}/I_{OL} \geq \pm 20mA$.

SU00011

Figure 18. Float Waveform

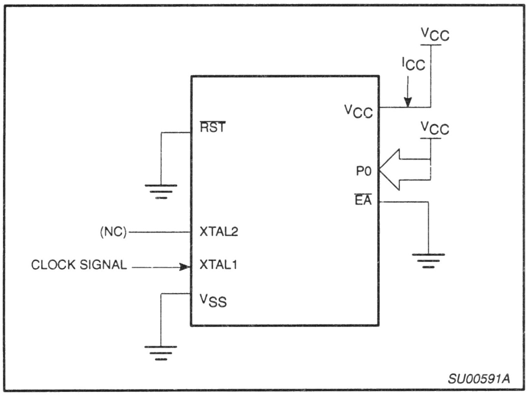

VCC
ICC
VCC
RST
VCC
P0
EA
(NC)
XTAL2
CLOCK SIGNAL
XTAL1
VSS
SU00591A

Figure 19. I_{CC} **Test Condition, Active Mode**
All other pins are disconnected

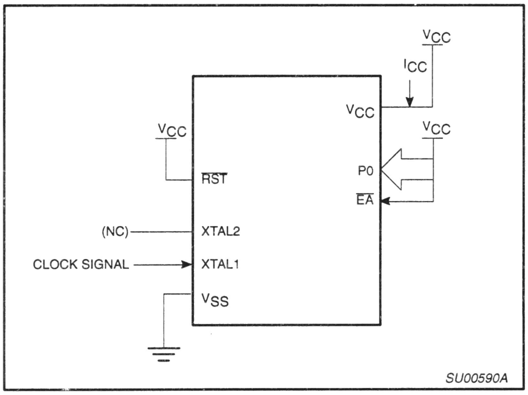

VCC
ICC
VCC
VCC
VCC
RST
P0
EA
(NC)
XTAL2
CLOCK SIGNAL
XTAL1
VSS
SU00590A

Figure 20. I_{CC} **Test Condition, Idle Mode**
All other pins are disconnected

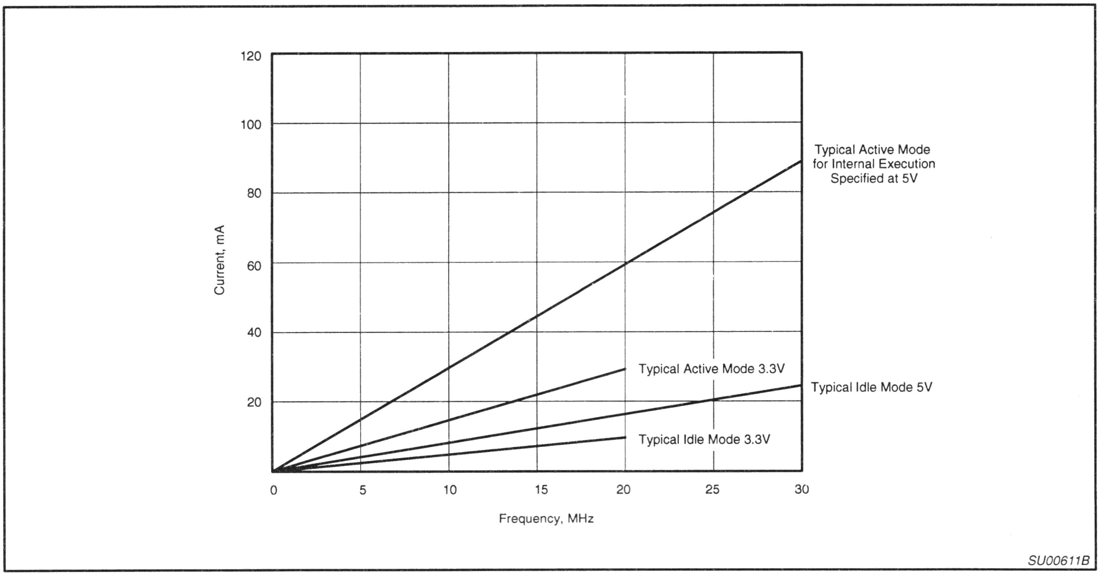


Figure 21. I_{CC} **vs. Frequency**
Valid only within frequency specification of the device under test.

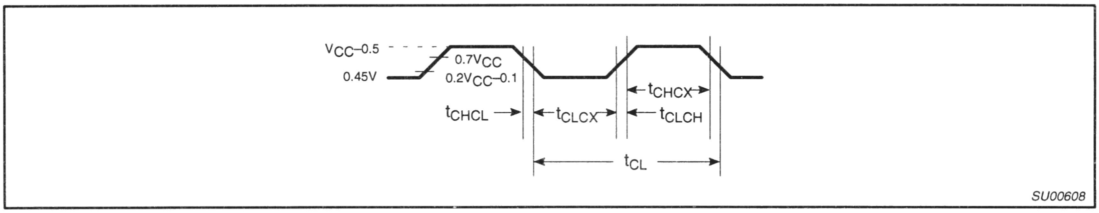

VCC–0.5
0.45V
0.7VCC
0.2VCC–0.1
tCHCL
tCLCX
tCHCX
tCLCH
tCL
SU00608

Figure 22. Clock Signal Waveform for I_{CC} **Tests in Active and Idle Modes**
$t_{CLCH} = t_{CHCL} = 5ns$

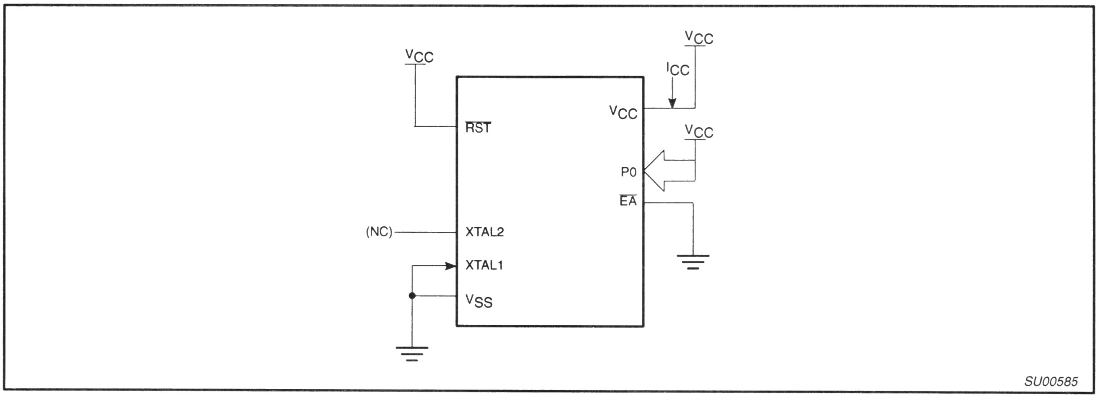


Figure 23. I_{CC} Test Condition, Power Down Mode
All other pins are disconnected. V_{CC}=2V to 5.5V

EPROM CHARACTERISTICS

The XA-G3 is programmed by using a modified Improved Quick-Pulse Programming™ algorithm. This algorithm is essentially the same as that used by the later 80C51 family EPROM parts. However different pins are used for many programming functions.

The XA-G3 contains three signature bytes that can be read and used by an EPROM programming system to identify the device. The signature bytes identify the device as an XA-G3 manufactured by Philips.

Table 5 shows the logic levels for reading the signature byte, and for programming the code memory and the security bits. The circuit configuration and waveforms for quick-pulse programming are shown in Figures 24. Figure 26 shows the circuit configuration for normal code memory verification.

Quick-Pulse Programming

The setup for microcontroller quick-pulse programming is shown in Figure 24. Note that the XA-G3 is running with a 3.5 to 12MHz oscillator. The reason the oscillator needs to be running is that the device is executing internal address and program data transfers.

The address of the EPROM location to be programmed is applied to ports 2 and 3, as shown in Figure 24. The code byte to be programmed into that location is applied to port 0. RST, $\overline{\text{PSEN}}$ and pins of port 1 specified in Table 5 are held at the 'Program Code Data' levels indicated in Table 5. The ALE/$\overline{\text{PROG}}$ is pulsed low 5 times as shown in Figure 25.

To program the security bits, repeat the 5 pulse programming sequence using the 'Pgm Security Bit' levels. After one security bit is programmed, further programming of the code memory and encryption table is disabled. However, the other security bits can still be programmed.

Note that the $\overline{\text{EA}}$/V_{PP} pin must not be allowed to go above the maximum specified V_{PP} level for any amount of time. Even a narrow glitch above that voltage can cause permanent damage to the device. The V_{PP} source should be well regulated and free of glitches and overshoot.

Program Verification

If security bits 2 and 3 have not been programmed, the on-chip program memory can be read out for program verification. The address of the program memory locations to be read is applied to ports 2 and 3 as shown in Figure 26. The other pins are held at the 'Verify Code Data' levels indicated in Table 5. The contents of the address location will be emitted on port 0.

Reading the Signature Bytes

The signature bytes are read by the same procedure as a normal verification of locations 030H, 031H, and 060H except that P1.2 and P1.3 need to be pulled to a logic low. The values are:

(030H) = 15H indicates manufactured by Philips
(031H) = EAH indicates XA architecture
(060H) = 01H indicates XA-G3

Program/Verify Algorithms

Any algorithm in agreement with the conditions listed in Table 5, and which satisfies the timing specifications, is suitable.

Erasure Characteristics

Erasure of the EPROM begins to occur when the chip is exposed to light with wavelengths shorter than approximately 4,000 angstroms. Since sunlight and fluorescent lighting have wavelengths in this range, exposure to these light sources over an extended time (about 1 week in sunlight, or 3 years in room level fluorescent lighting) could cause inadvertent erasure. **For this and secondary effects, it is recommended that an opaque label be placed over the window.** For elevated temperature or environments where solvents are being used, apply Kapton tape Fluorglas part number 2345–5, or equivalent.

The recommended erasure procedure is exposure to ultraviolet light (at 2537 angstroms) to an integrated dose of at least 15W-s/cm^2. Exposing the EPROM to an ultraviolet lamp of 12,000μW/cm^2 rating for 90 to 120 minutes, at a distance of about 1 inch, should be sufficient.

Erasure leaves the array in an all 1s state.

™Trademark phrase of Intel Corporation.

Security Bits

With none of the security bits programmed the code in the program memory can be verified. When only security bit 1 (see Table 5) is programmed, MOVC instructions executed from external program memory are disabled from fetching code bytes from the internal memory. All further programming of the EPROM is disabled. When security bits 1 and 2 are programmed, in addition to the above, verify mode is disabled. When all three security bits are programmed, all of the conditions above apply and all external program memory execution is disabled. (See Table 6)

Table 5. EPROM Programming Modes

MODE	$\overline{RST}$	$\overline{PSEN}$	ALE/$\overline{PROG}$	$\overline{EA}$/V_{PP}	P1.0	P1.1	P1.2	P1.3	P1.4
Read signature	0	0	1	1	0	0	0	0	0
Program code data	0	0	0*	V_{PP}	0	1	1	1	1
Verify code data	0	0	1	1	0	0	1	1	0
Pgm security bit 1	0	0	0*	V_{PP}	1	1	1	1	1
Pgm security bit 2	0	0	0*	V_{PP}	1	1	0	0	1
Pgm security bit 3	0	0	0*	V_{PP}	1	0	1	0	1
Verify security bits	0	0	1	1	0	0	0	1	0

NOTES:
1. '0' = Valid low for that pin, '1' = valid high for that pin.
2. V_{PP} = 12.75V ±0.25V.
3. V_{CC} = 5V±10% during programming and verification.

* ALE/$\overline{PROG}$ receives 5 programming pulses (only for user array; 25 pulses for encryption or security bits) while V_{PP} is held at 12.75V. Each programming pulse is low for 100µs (±10µs) and high for a minimum of 10µs.

Table 6. Program Security Bits

PROGRAM LOCK BITS				
	SB1	SB2	SB3	PROTECTION DESCRIPTION
1	U	U	U	No Program Security features enabled.
2	P	U	U	MOVC instructions executed from external program memory are disabled from fetching code bytes from internal memory and further programming of the EPROM is disabled.
3	P	P	U	Same as 2, also verify is disabled.
4	P	P	P	Same as 3, external execution is disabled. Internal data RAM is not accessible.

NOTES:
1. P – programmed. U – unprogrammed.
2. Any other combination of the security bits is not defined.

ROM CODE SUBMISSION

When submitting ROM code for the XA-G3, the following must be specified:
1. 32k byte user ROM data
2. ROM security bits.
3. Watchdog configuration

ADDRESS	CONTENT	BIT(S)	COMMENT
0000H to 7FFFH	DATA	7:0	User ROM Data
8020H	SEC	0	ROM Security Bit 1
8020H	SEC	1	ROM Security Bit 2 0 = enable security 1 = disable security
8020H	SEC	3	ROM Security Bit 3 0 = enable security 1 = disable security

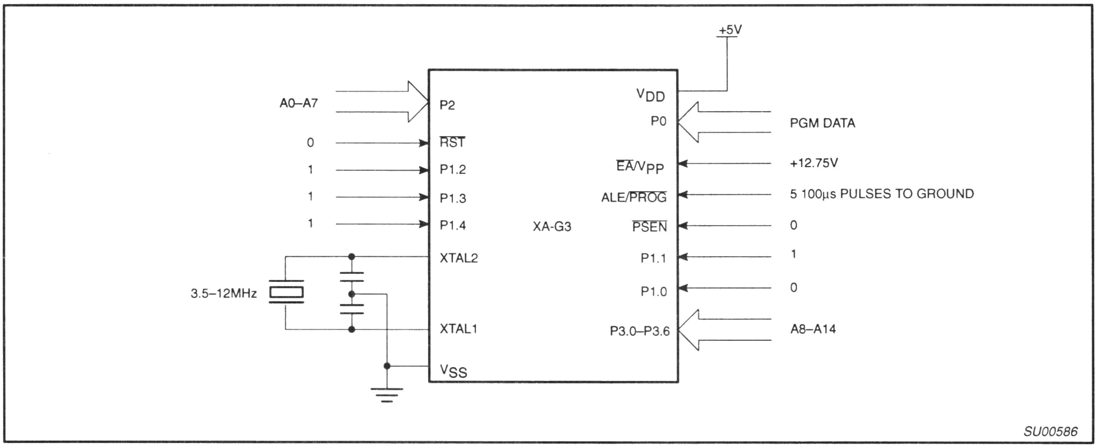

+5V
A0–A7
P2
VDD
P0
PGM DATA
0
RST
1
P1.2
EA/VPP
+12.75V
1
P1.3
ALE/PROG
5 100μs PULSES TO GROUND
1
P1.4
XA-G3
PSEN
0
XTAL2
P1.1
1
3.5–12MHz
P1.0
0
XTAL1
P3.0–P3.6
A8–A14
VSS
SU00586

Figure 24. Programming Configuration for XA-G3

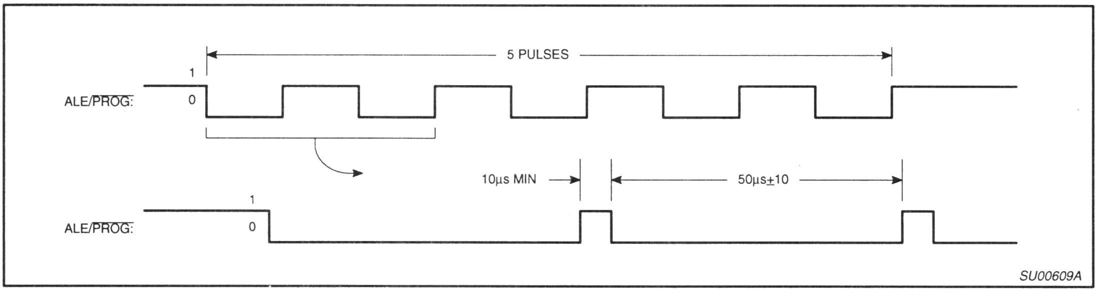

5 PULSES
1
ALE/PROG:
0
10μs MIN
50μs±10
1
ALE/PROG:
0
SU00609A

Figure 25. PROG Waveform

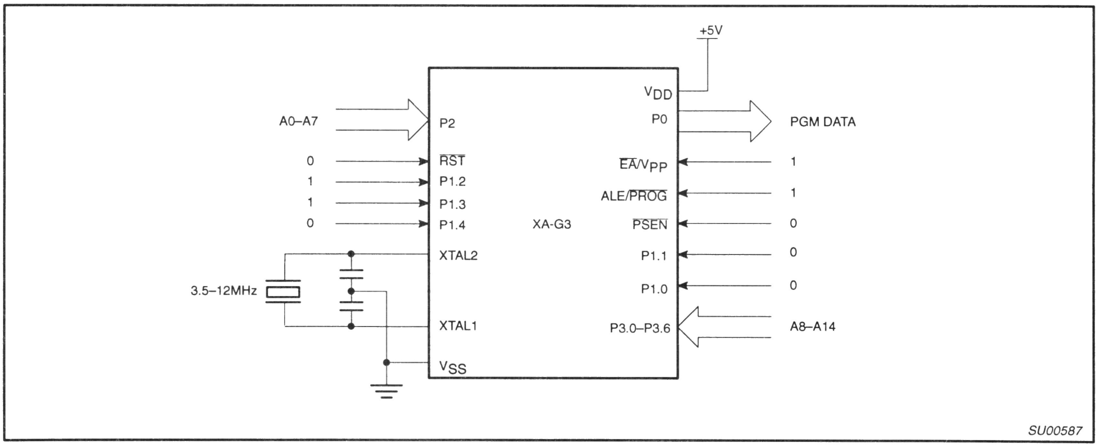

+5V
A0–A7
P2
VDD
P0
PGM DATA
0
RST
1
P1.2
EA/VPP
1
1
P1.3
ALE/PROG
1
0
P1.4
XA-G3
PSEN
0
XTAL2
P1.1
0
3.5–12MHz
P1.0
0
XTAL1
P3.0–P3.6
A8–A14
VSS
SU00587

Figure 26. Program Verification for XA-G3

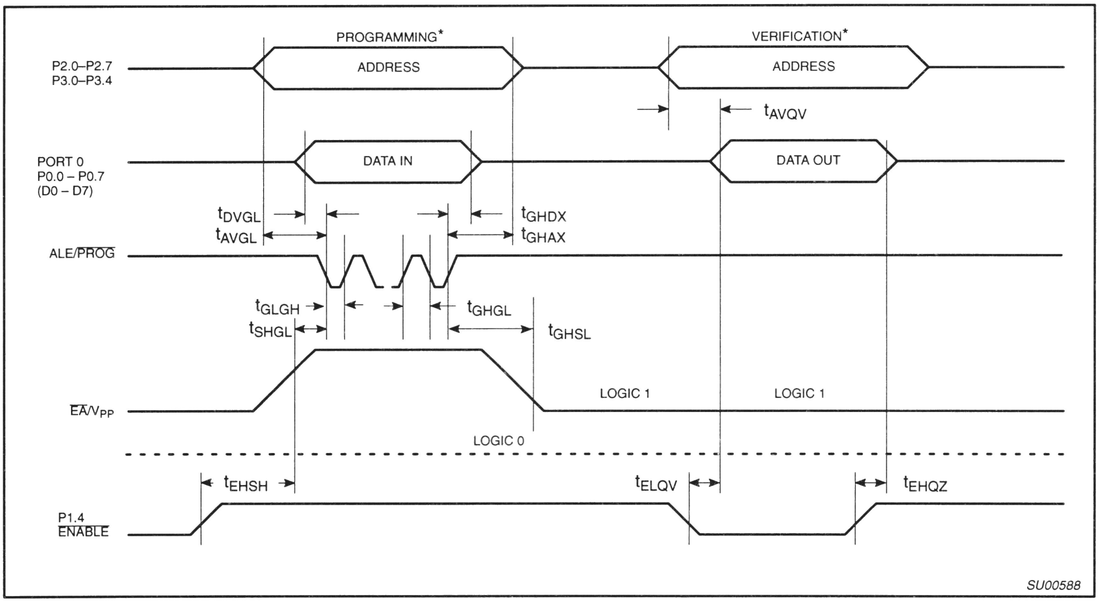
EPROM PROGRAMMING AND VERIFICATION CHARACTERISTICS

T_{amb} = 21°C to +27°C, V_{DD} = 5V±10%, V_{SS} = 0V (See Figure 27)

SYMBOL	PARAMETER	MIN	MAX	UNIT
V_{PP}	Programming supply voltage	12.5	13.0	V
I_{PP}	Programming supply current		50 [1]	mA
$1/t_{CL}$	Oscillator frequency	3.5	12	MHz
t_{AVGL}	Address setup to $\overline{\text{PROG}}$ low	$48t_{CL}$		
t_{GHAX}	Address hold after $\overline{\text{PROG}}$	$48t_{CL}$		
t_{DVGL}	Data setup to $\overline{\text{PROG}}$ low	$48t_{CL}$		
t_{GHDX}	Data hold after $\overline{\text{PROG}}$	$48t_{CL}$		
t_{EHSH}	P2.7 ($\overline{\text{ENABLE}}$) high to V_{PP}	$48t_{CL}$		
t_{SHGL}	V_{PP} setup to $\overline{\text{PROG}}$ low	10		μs
t_{GHSL}	V_{PP} hold after $\overline{\text{PROG}}$	10		μs
t_{GLGH}	$\overline{\text{PROG}}$ width	40	60	μs
t_{AVQV}	Address to data valid		$48t_{CL}$	
t_{ELQV}	$\overline{\text{ENABLE}}$ low to data valid		$48t_{CL}$	
t_{EHQZ}	Data float after $\overline{\text{ENABLE}}$	0	$48t_{CL}$	
t_{GHGL}	$\overline{\text{PROG}}$ high to $\overline{\text{PROG}}$ low	10		μs

NOTE:

1. Not tested.


PROGRAMMING*
VERIFICATION*
P2.0–P2.7
P3.0–P3.4
ADDRESS
ADDRESS
t_{AVQV}
PORT 0
P0.0 – P0.7
(D0 – D7)
DATA IN
DATA OUT
t_{DVGL}
t_{AVGL}
t_{GHDX}
t_{GHAX}
ALE/$\overline{\text{PROG}}$
t_{GLGH}
t_{SHGL}
t_{GHGL}
t_{GHSL}
$\overline{\text{EA}}/V_{PP}$
LOGIC 1
LOGIC 1
LOGIC 0
t_{EHSH}
t_{ELQV}
t_{EHQZ}
P1.4
$\overline{\text{ENABLE}}$
SU00588

NOTE:

* FOR PROGRAMMING CONDITIONS SEE FIGURE 25.
FOR VERIFICATION CONDITIONS SEE FIGURE 26.

Figure 27. EPROM Programming and Verification

Section 4

Future Derivatives

CONTENTS

XA-C3	16-bit microcontroller with on-chip CAN	407
XA-S3	Single-chip 16-bit microcontroller	408

16-bit microcontroller with on-chip CAN XA-C3

DESCRIPTION

The XA-C3 device is a member of Philips' 80C51 XA (eXtended Architecture) family of high performance 16-bit single-chip microcontrollers, and is intended for industrial control applications.

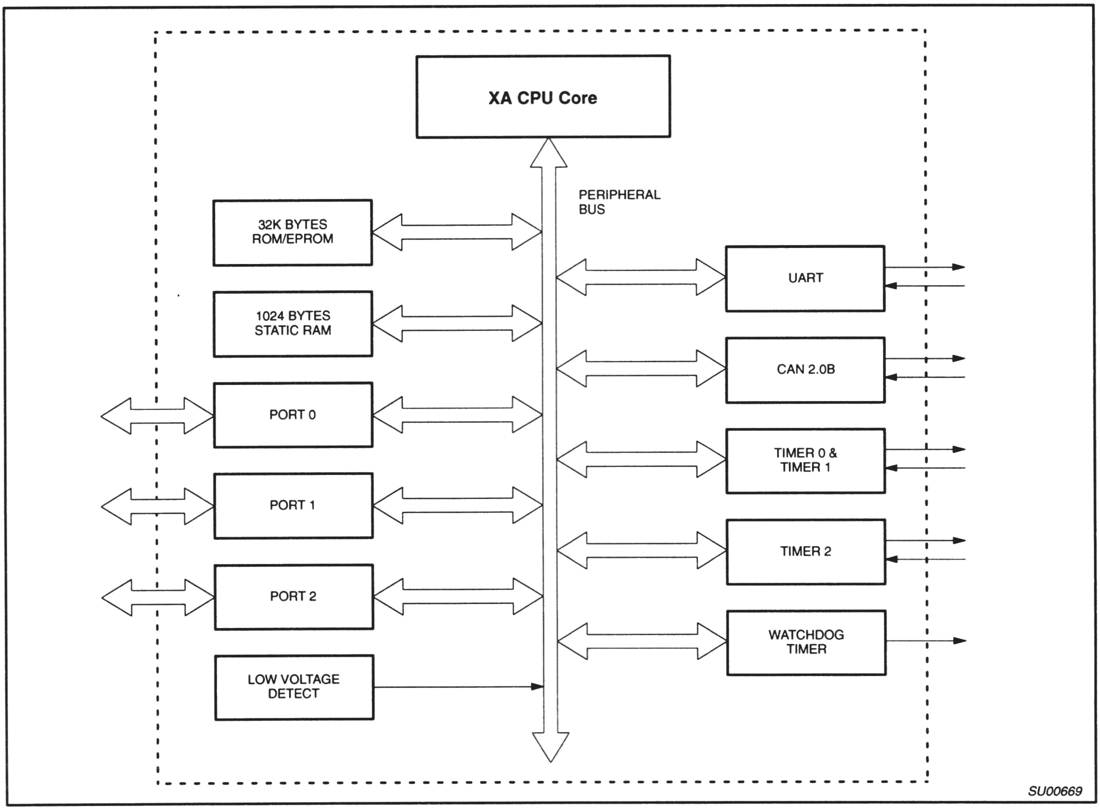
The XA-C3 device supports the full Controller Area Network (CAN) 2.0B. It supports both 11-bit and 29-bit identifiers (ID) at up to 1Mbit/s data rate. It is further optimized for DeviceNet™ applications.

The performance of the XA architecture supports the comprehensive bit-oriented operations of the 80C51 while incorporating support for multi-tasking operating systems and high-level languages such as C. The speed of the XA architecture, at 10 to 100 times that of the 80C51, gives designers an easy path to truly high performance embedded control, while maintaining great flexibility to adapt software to specific requirements.

Specific Features of the XA-C3

- 2.7V to 5.5V operation
- 32K bytes of on-chip EPROM/ROM program memory
- 1024 bytes of on-chip data RAM
- CAN block supporting full CAN2.0B, with 11-/29-bit ID and up to 1Mbit/s
- Three standard counter/timers with enhanced features (equivalent to 80C51 T0, T1, and T2) with outputs
- Watchdog timer with output
- 1 UART
- Low voltage detect
- Three 8-bit I/O ports with 4 programmable output configurations
- EPROM/OTP versions can be programmed in circuit
- 25MHz operating frequency at 4.5 – 5.5V V_{CC} over commercial operating conditions; 16MHz at 2.7V – 3.6V V_{CC}
- 40-pin DIP, 44-pin PLCC, and 44-pin QFP packages

BLOCK DIAGRAM



DeviceNet™ is a trademark of Open DeviceNet Vendor Association (OVDA).

Single-chip 16-bit microcontroller

XA-S3

DESCRIPTION

The XA-S3 device is a member of Philips' 80C51 XA (eXtended Architecture) family of high performance 16-bit single-chip general purpose microcontrollers.

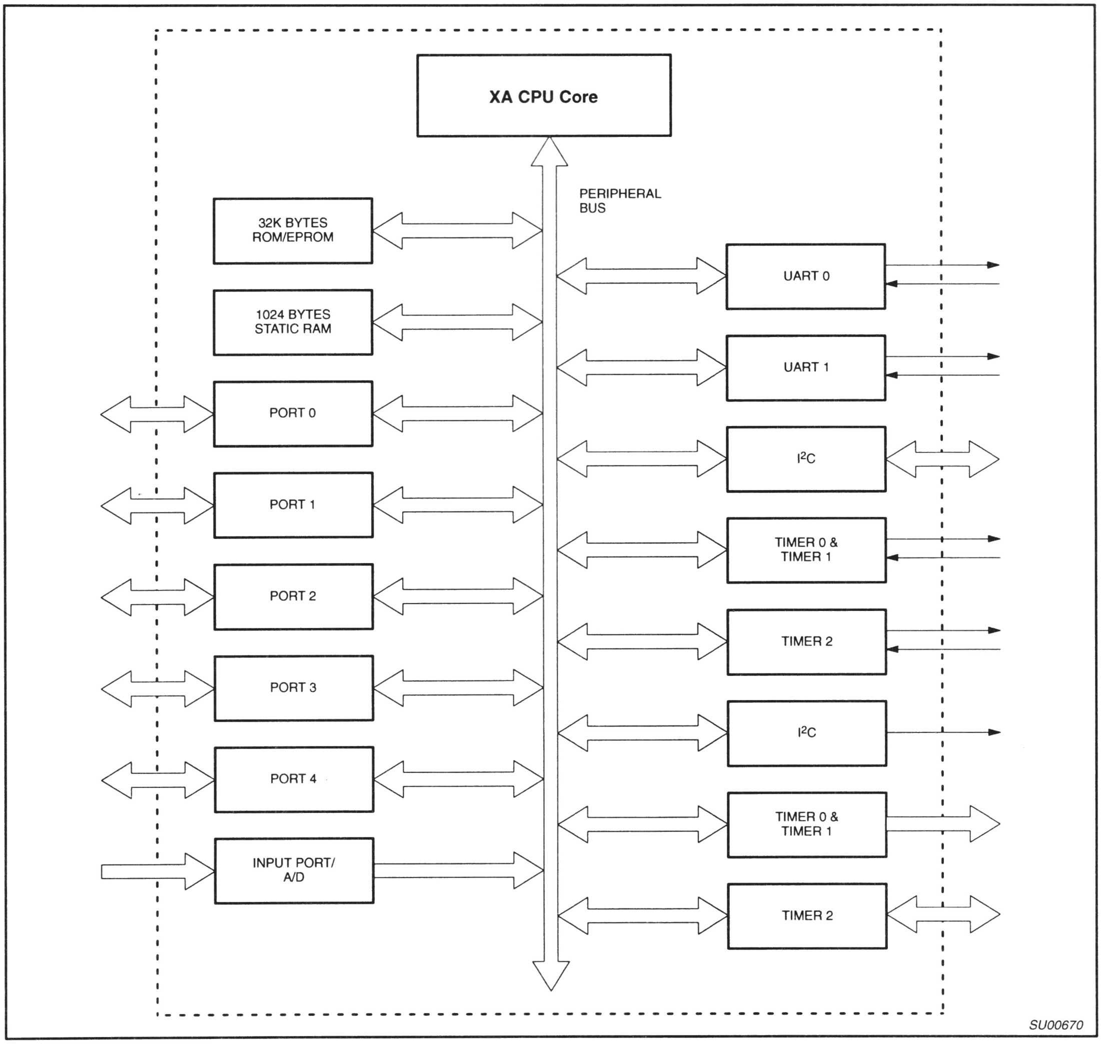
The XA-S3 device combines many powerful peripherals on chip. With its dual-channel Universal Peripheral Interface (UPI), high performance A/D converters, timers/counters, watchdog, Programmable Counter Array (PCA), I^{2}C interface, UARTs and multiple general purpose I/O ports, it is optimized for general multipurpose high performance embedded control functions.

The performance of the XA architecture supports the comprehensive bit-oriented operations of the 80C51 while incorporating support for multi-tasking operating systems and high-level languages such as C. The speed of the XA architecture, at 10 to 100 times that of the 80C51, gives designers an easy path to truly high performance embedded control, while maintaining great flexibility to adapt software to specific requirements. the performance of the XA architecture, together with the powerful and extensive peripherals in the XA-552, provide a powerful alternative for embedded applications.

Specific Features of the XA-S3

- 2.7V to 5.5V operation
- 32K bytes of on-chip EPROM/ROM program memory
- 1024 bytes of on-chip data RAM
- Dual-channel 16-bit Universal Peripheral Interface (UPI)
- High performance 8-channel 8-bit A/D converter
- Three standard counter/timers with enhanced features (equivalent to 80C51 T0, T1, and T2) with outputs
- Watchdog timer with output
- 5-channel 16-bit Programmable Counter Array (PCA)
- I^{2}C-bus serial I/O port with byte-oriented master and slave functions and software address recognition
- Two enhanced UARTs with separate baud rate generators
- Five 8-bit I/O ports with 4 programmable output configurations, plus one 8-bit input port shared with analog inputs
- EPROM/OTP versions can be programmed in circuit
- 25MHz operating frequency at 4.5 – 5.5V V_{CC} over commercial operating conditions; 16MHz at 2.7V – 3.6V V_{CC}
- 80-pin QFP packages

BLOCK DIAGRAM


XA CPU Core
PERIPHERAL BUS
32K BYTES ROM/EPROM
1024 BYTES STATIC RAM
PORT 0
PORT 1
PORT 2
PORT 3
PORT 4
INPUT PORT/ A/D
UART 0
UART 1
I2C
TIMER 0 & TIMER 1
TIMER 2
I2C
TIMER 0 & TIMER 1
TIMER 2
SU00670

Section 5

Application Notes

CONTENTS

AN700 Digital filtering using XA Revision 0.11 413
AN701 SP floating point math with XA 418
AN702 High level language support in XA 441
AN703 XA benchmark versus the architectures 68000, 80C196, and 80C51 445
AN704 An upward migration path for the 80C51: the Philips XA architecture 469

Digital filtering using XA Revision 0.11

AN700

Author: Santanu Roy, MCO Applications Group, Sunnyvale, California

SUMMARY

This report describes a method of implementation of FIR filters using Philips XA microcontroller. Appended with this application note is a generic routine that could be used to implement a N-point FIR filter.

INTRODUCTION

The term "digital filter" refers to the computational process or algorithm by which a digital signal or sequence of numbers (acting as input) is transformed into a second sequence of numbers termed the output digital signal. Digital filters involve signals in the digital domain (discrete-time signals) and are used extensively in applications, such as digital image processing, pattern recognition, and spectral analysis.

Digital Signal Processing (DSP) is concerned with the representation of signals (and information they contain) by sequences of numbers and with the transformation or processing of such signal representations by numeric computational procedures. In order to be considered a DSP microcontroller, a part must be able to quickly multiply two values, and add the result to an accumulator register. this is a minimum requirement. "Quickly" implies MAC (Multiply and Accumulate). Typically6, the multiply and accumulate path operates on 16-bit values with a 32-bit result. Figure 1 shows a typical Digital Signal Processing hardware used in digital filtering.

Although XA does not have a hardware MAC unit for DSP applications, it is quite suitable for some DSP applications, due to its relatively high computational power, and high I/O throughput. This application note is intended to demonstrate such DSP power of the XA through implementation of FIR and IIR digital filters. It is to be noted, though, that this application note is not intended as a learning tool for DSP, so it is assumed that the reader is familiar with DSP and filtering basics.

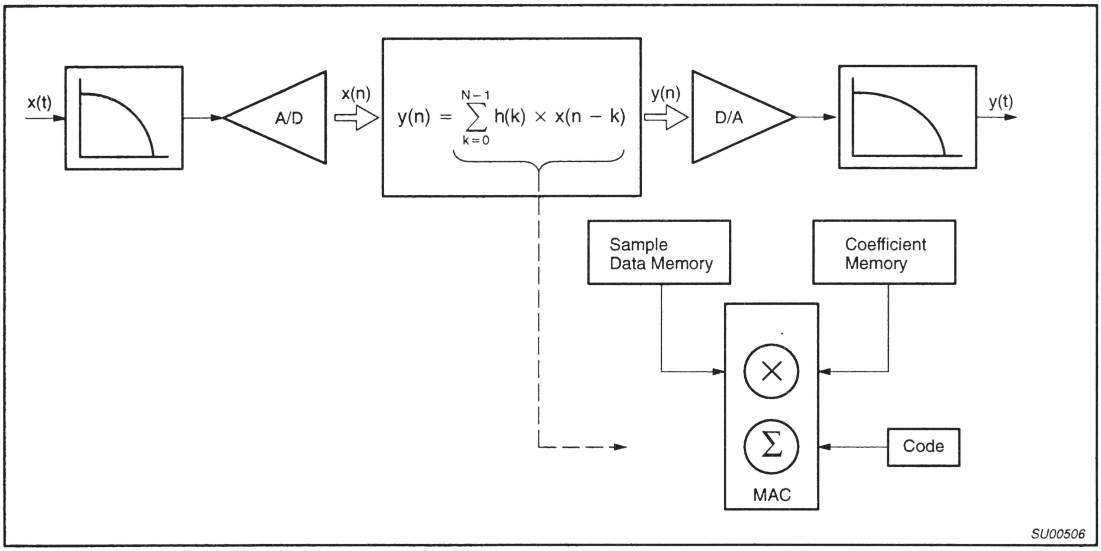


Figure 1. Typical DSP Hardware

Filter Algorithms

For a large variety of applications, digital filters are usually based on the following relationship between the filter input sequence x(n) and filter output sequence y(n);

$$y(n) = \sum_{k=0}^{N} a_k \times y(n-k) + \sum_{k=0}^{M} b_k \times x(n-k) \qquad (1)$$

where a_k and b_k represent constant coefficients and N and M represent the number of input samples.

Equation (1) is referred to as a linear constant coefficient difference equation. Two classes of filters can be represented by such equations:

4. Finite Impulse Response (FIR) filters, and
5. Infinite Impulse Response (IIR) filters.

This applications note describes the implementation of the FIR class of digital filters on the XA.

FIR Filters

FIR filters are preferred in lower order solutions, and since they do not employ feedback (output values used in the calculation of newer output values), they exhibit naturally bounded response. They are simpler to implement, and require one RAM location and one coefficient for each order.

For FIR filters, all of the a_k in equation (1) is zero. Therefore (1) reduces to:

$$y(n) = \sum_{k=0}^{M} b_k \times x(n-k) \qquad (2)$$

As a result, the output of an FIR filter is simply a finite length weighted sum of the present and previous inputs to the filter. If the unit-sample response of the filter is denoted as h(n), then from (2), it is seen that h(n) = b(n). Therefore, (2) is sometimes written as:

$$y(n) = \sum_{k=0}^{N-1} h(k) \times x(n-k) \qquad (3)$$

where N = length of the filter = M+1.

Digital Filter Implementation

As described above, a digital filter (FIR or IIR) could then be implemented by multiplying a vector of sampled signals with another vector of constants (coefficients) and adding the results to a register. The vectors involved in the filter process are derived from transformation of an S domain transfer function into the sampled Z domain.

The Multiply-Accumulate (MAC) Function

The MAC speed applies both to finite impulse response (FIR) and finite impulse response (IIR) filters. The complexity of the filter response dictates the number MAC operations required per sample period.

A multiply-accumulate step performs the following:

- Read a 16-bit sample data (pointed to by a register)
- Increment the sample data pointer by 2
- Read a 16-bit coefficient (pointed to by another register)
- Increment the coefficient register pointer by 2
- Sign Multiply (16-bit) data and coefficient to yield a 32-bit result
- Add the result to the contents of a 32-bit register pair for accumulate.

This accumulator should be initialized to zero before calculating each output. It is assumed that the algorithm cannot overflow the accumulator, either by reducing significant bits of samples and/or constants or the number of accumulations.

All these above MAC operations take place in addition to a buffer management routine that maintains an updated database for the filters samples, and system coefficients.

Buffer Management

In order to effectively perform the task of buffer management, the processor should be able to quickly "shift" data (or pointers) in a data array which contains a series of input samples. New data is going in, and oldest sample is disposed off.

There are few ways to maintain and manage this database. They are as follows:

1. Linear Buffer

New Sample	x(n)	
	x(n–1)	↑
	x(n–2)	Increasing Address
	x(n–3)	
Lost	x(n–4)	

Linear buffer management requires the data to physically move down towards the oldest sample, then the newest sample is written into the top (FIFO style).

2. Circular Buffer

	x(n–1)	
	x(n)	
*1 –>	x(n–4)	
	x(n–3)	<– *2
	x(n–2)	

*1 At the beginning of filter pass, input pointer points to the oldest (n–4) sample, new sample is stored there.
*2 At the end of the filter pass, pointer now points to the next oldest sample (n–3).

Circular buffer management requires a test to make sure a buffer pointer increment does not move the pointer beyond the "tail" (end) of the buffer. If so, the pointer must be reset to the "head" (beginning) of the buffer.

Selecting one approach versus another depends mainly on the overhead involved with this task over the plain Multiply-Accumulate and loop control operations, and may vary based on the processor architecture, storage access time and other factors.

MAC Implementation on the XA

An efficient loop for memory mapped vectors is presented below. The loop entry is at an even address, to reduce the fetch overhead after branch to beginning of the loop. Arrays are accessed using the indirect-autoincrement addressing mode.

```
.incld fir.h
  MAC_LOOP:
       mov.w    R3, [R1+]     ; read sample vector entry
       mov.w    R4, [R2+]     ; read coefficient vector entry
       mul.W    R3, R4        ; multiply
       add      R5, R3        ; accumulate into
       addc     R6, R4        ; a 32 bit register pair(R5:R6)
                              ; this serves as the RRP
       djnz     R0, MAC_LOOP ;
                             ;decrement loop counter and
                             ;branch to MAC_LOOP
```

The loop contains 13 bytes and takes 32 clocks (including branch penalty) per iteration (1.6μS at 20MHz and 107μS at 30.0MHz).

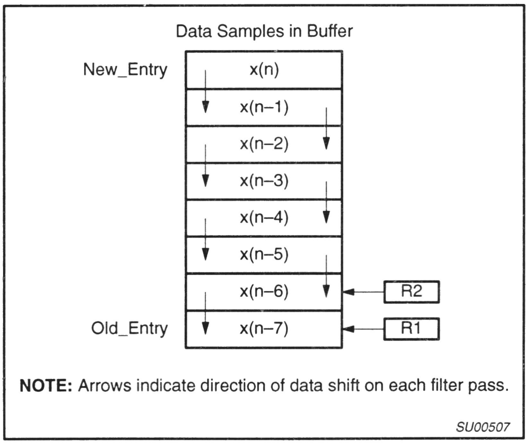
The following section analyzes the digital filter performance, including initialization, I/O, MAC operations and sample vector buffer management.

An N-Point FIR Filter Implementation on XA

The FIR filter maintains a list of a fixed number N of recent samples. At each iteration, a new sample is taken, replacing the oldest sample on the list. This list represents a sampled vector. It is then multiplied by an N constant's vector to yield the current output.

As mentioned earlier, there are 2 register pointers fetching data samples and coefficient and feeding it to the ALU for 16-bit signed multiply with the 32-bit result being added to the MAC result register pair (RRP). In addition, a buffer management routine updates the sample data buffer each sample period.

The following sample codes show the mechanism for running filters on successive samples. It reflects the simplest data structures and list management, to simulate an output of a high level compiler.



NOTE: Arrows indicate direction of data shift on each filter pass.

Figure 2. Buffer Management for FIR Filter

FIR Algorithm in XZ

```
;Preliminary initialization for first filter pass:
.incldfir.h
  Start_FIR:
    mov  R0, #N-1           ; N = number of entries in the list
                            ; = loop counter
    mov  R1, #Old_Entry     ; compiler uses 2 pointers
    mov  R2, #Old_Entry+2   ;

Shft_Smpl:
    mov.w   [R1+], [R2+]
    djnz    R0, Shft_Smpl
                            ; - SAMPLE FROM A/D PORT
    mov     R0, A2D         ; input from port
    and     R0, #mask       ; mask upper bits (for N-bit A/D)
    mov     New_Entry, R0   ; add to list

Mac_init:                   ; MULTIPLY ACCUMULATE
    mov     R0, #N          ; N = number of entries in the list
                            ; = loop counter
    mov     R1, #Old_Entry  ; pointer to sample vector
    mov     R2, #Coef_Entry ; pointer to coefficient vector
    xor     R5, R5          ; zero to accumulator
    xor     R6, R6          ; zero to accumulator

MAC_LOOP:
    mov.w   R3, [R1+]       ; read sample from list to reg
    mov.w   R4, [R2+]       ; read constant from list to reg
    mul.w   R3, R4          ; multiply
    add     R5, R3          ; accumulate in RRP
    addc    R6, R4          ; complete 32 bit add
    djnz    R0, MAC_LOOP

ACC_corr:                   ; - NORMALIZE RESULT BY SHIFTING
    asl     R5, #norm**     ; correction for non-significant LSBs
                            ; for eight 10 bit samples and 16 bit
                            ; constants, #norm=3
                            ; i.e. take only most significant 29 bits of the result
                            ; [16+10 + 3 (for 8 iterations)]
                            ; - OUTPUT TO D2A PORT
    mov     DAC, R6         ; send to DAC

        A total of 62 bytes and 370 clocks for this FIR algorithm.
```

** For N=8, 10 bit A/D, 16 bit filter coefficients; 8, 12, 16 bits clock very similar performance.

Total time for an 8-point filter at 20MHz is 19.0 microseconds and 12.7 microseconds at 30MHz. This would translate to a maximum sampling rate of 52KHz at 20MHz and 78KHz at 30MHz clock. If this filter algorithm is interrupt driven, then additional 20 clocks would be required for latency, which would then translate to 50KHz maximum sampling rate at 20MHz and 75KHz at 30MHz. this puts the XA in the bandwidth of Audio Signal Processing (44.1KHz) applications.

NOTES:

1. The above FIR algorithms are assembled with "asmxa rev 1.4", the first XA absolute assembler for verification. It is to be noted in this context, that this assembler is a beta-site tool and still under evaluation. The syntax used in the assembler might be subjected to change. The functionality of the code is not checked at this stage using any simulator or ICE.
2. It is possible in the above MAC operation to extend the length of the accumulator to accommodate more iterations and higher precision (greater than 10-bit A/D) sample values with some additional overhead, e.g., using 'ADDC Rn, R6H", etc., after the 32-bit accumulate, where Rn is a byte-size register to increase the length of the accumulator to accommodate more accumulations and higher precision (greater than 10-bit A/D) sample values.

Digital filtering using XA Revision 0.11

Author's Note

All addresses and constants assumed 16 bit for generality. Performance is calculated for a work-aligned branch targets which is mandated in the XA architecture for performance reasons. Misalignment will result in addition of NOPs by the assembler causing penalty in both code density and execution times. It is also to be mentioned that this is not the fastest executable code for the XA. A good programmer can combine the two loops into one, and data can be kept in registers. For low order filter implementation, code can be written in-line, and can utilize direct addressing mode for samples array.

This code was written in a way that reflects minimum expected optimization form a compiler (local loop optimization only), and it shows the expected speed for code written in a high level language, without rewriting routines in assembly language. also, this is not the ultimate performance for the XA architecture. The register banks can be used to store coefficients and samples, resulting in slightly faster execution time.

Author's Acknowledgement

The author recognizes the following Philips Semiconductors XA team members for their review and inputs on this article:

Greg Goodhue, Ori Mizrahi-Shalom, and Ata Khan.

References

XA User Guide — Philips Semiconductors
Digital Signal Processing — Rosenbaum

SP floating point math with XA

AN701

Author: Santanu Roy, MCO Applications Group, Sunnyvale, California

IEEE SINGLE PRECISION FLOATING POINT ARITHMETIC WITH XA

Introduction

This application note is intended to implement Single Precision Floating Point Arithmetic package using the new Philips Semiconductors XA microcontroller. The goal is to have this package as a part of the run-time math library for the XA when the cross-compiler is developed. The package is based upon the IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985). This package, however, is not a conforming implementation of the said standard. The differences between the XA implementation and the standard are listed later in this report. Also, this package does not include routines for conversion between Integers to Floating Point and vice versa.

The following four standard Single Precision (SP) arithmetic operations have been implemented in this package:

1.	**FPADD**	Addition of two SP floating point numbers.
2.	**FPSUB**	Subtraction of two SP floating point numbers.
3.	**FPMUL**	Multiplication of two SP floating point numbers.
4.	**FPDIV**	Division of two SP floating point numbers.

The following section discusses the representation of FLP numbers. Then the differences between the XA implementation and the IEEE standard are described. This is followed by a description of the algorithms used in the computations. Appendix A is a user reference section for converting a floating point number into IEEE format and Appendix B is a listing of the code.

Note that this application note assumes that the reader is familiar with the IEEE Binary Floating-Point standard.

IEEE Floating Point Formats

The basic format sizes for floating-point numbers, 32 bits and 64 bits, were selected for efficient calculation of array elements in byte-addressable memories. For the 32-bit format, precision was deemed the most important criterion, hence the choice of radix 2 instead of octal or hexadecimal. Other characteristics include not representing the leading significand bit in normalized numbers, a minimally acceptable exponent range which uses 8 bits, and exponent bias which allows the reciprocal of all normalized numbers to be represented without overflow. For the 64-bit format, the main consideration was range.

Representation of FLP Number

The IEEE binary floating point number is represented in the following format:

$FP = \pm \text{Significand} \times \text{Base}^{\text{Characteristic}}$

The specification of a binary FP number involves two parts:
A Significand or Mantissa
and a Characteristic or Exponent.

The Mantissa is signed fixed point number and the Exponent is a signed integer. Mantissa or Significand is that component of a binary FLP number which consists of an explicit or implicit leading bit to the left of its binary point and a fraction field to the right of the binary point. Exponent signifies the power to which 2 is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent. The IEEE standard specifies that a single precision Floating Point number should be represented in 32 bits as shown in Figure 1.

SIGN 1-bit	EXPONENT 8-bits	MANTISSA 23-bits

Figure 1.

The significance of each of these fields is as follows:

1. **SIGN** — This 1–bit field is the sign of the Mantissa. '0' indicates a *positive* and '1' indicates a *negative* number.
2. **EXPONENT** — This is a 8–bit field. The width of this field determines the range of the FP number. The exponent is represented as a biased value with a bias of 127 decimal. The bias value is added to exponents in order to keep them always positive and is represented by

   $2^{n-1} - 1,$

   where n = number of bits in the binary exponent.

3. **MANTISSA** — This is a 23-bit field representing the fractional part. The width of this field determines the precision for the FP number. For normalized FP numbers (see below), a MSB of '1' is assumed and not represented. Thus, for normalized numbers, the value of the mantissa is 1.Mantissa. This provides an effective precision of 24–bits for the mantissa.

   If dealt with normalized numbers only (as the XA implementation does), then the MSB of the Mantissa need not be explicitly represented as per IEEE standard specification.The normalized significand lies in the range shown below.

   $1.0 < \text{Normalized Mantissa} < 2.0$

   Given the values of Sign, Exponent, and Mantissa, the value of the FP number is obtained as follows:

(i) If $0 < \text{Exp} < 255$, then

$FP = (-1)^{\text{SIGN}} \times 2^{\text{EXP} - 127} \times 1.\text{MANTISSA}$

(ii) If Exp = 0, then FP = 0

(iii) If Exp = 255, and Mantissa =/= 0, then FP = Invalid Number (NaN or Not a Number).

The above format for single precision binary FP numbers provides for the representation in the range -3.4×10^{38} to -1.75×10^{-38}, 0, and 1.75×10^{-38} to 3.4×10^{38}. The accuracy is between 7 and 8 decimal digits.

Differences with the IEEE Standards

The IEEE standard specifies a comprehensive list of operations and representations for FLP numbers. Since an implementation that fully conforms to this standard would lead to an excessive amount of overhead, a number of features in the standard were omitted. This section describes the differences between the implemented package and the standard.

1. **Omission of –0** — The IEEE standard requires that both + and – 0 be represented, and arithmetic carried out using both. The implementation does not represent –0.
2. **Omission of infinity arithmetic** — The IEEE standard provides for the representation of + and – infinity, and requires that valid arithmetic operations be carried out on infinity.
3. **Omission of Quiet NaN** — The IEEE standard provides for both Quiet and Signalling NaNs. A signalling NaN can be produced as

the result of overflow during an arithmetic operation. If the NaN is passed as input to further FLP routines, then these routines would produce another NaN as output. The routines will also set the Invalid Operation Flag, and call the user FLP error trap routine at address FPTRAP.

4. **Omission of denormalized numbers** — These are FLP numbers with a biased exponent, E of zero and non-zero mantissa F. Such denormalized numbers are useful in providing gradual underflow to 0. These are not represented in the XA implementation. Instead, if the result of a computation cannot be represented as a normalized number within the allowable exponent range, then an underflow is signalled, the result is set to 0, and the user FLP error trap routine at address FPTRAP is called.
5. **Omission of Inexact Result Exponent** — The IEEE standard requires that an Inexact Result Exception be signalled when the round result of an operation is not exact, or it overflows without an overflow trap. This feature is not provided.
6. **Biased Rounding to Nearest** — The IEEE standard requires that rounding to the nearest be provided as the default rounding mode. Further, the rounding is required to be unbiased. The XA implementation provides biased rounding to nearest only, e.g., suppose the result of an operation is .b1b2b3nnn and needs to be rounded to 3 binary digits. Then if nnn is 0YY, the round to nearest result is .b1b2b3. If nnn is 1YY, with at least one of the Y's being 1, then the result is .b1b2b3 + 0.001. Finally, if nnn is 100, it is a tie situation. In such a case, the IEEE standard requires that the rounded result be such that its LSB is 0. The XA implementation, on the other hand, will round the result in such a case to .b1b2b3 + 0.001.

DESCRIPTION OF ALGORITHMS

General Considerations

The XA implementation of the SP floating point package consists of a series of subroutines. The subroutines have been written and tested using Microsoft C however, not been tested with the XA C cross-compiler and also not optimized for code efficiency. The executable could be run under DOS in any IBM compatible PC. It is a menu driven routine that enables the user to select any of the 4 Floating Point routines. The menu also includes a "HELP" item designed to provide some standard SP floating point numbers, their operations and results as a quick reference.

The Arithmetic subroutines that compute F1 (Op) F2, where Op is +, –, *, or / expect that F1 and F2 are in IEEE format. Each of F1 and F2 consists of two 16-bit words organized as follows:

Fn–HI : Sign(1) Biased exponent(7) MSB of Mantissa(8)

Fn–LO : Least Significant word of Mantissa(16)

Exception Handling

The following types of exception can occur during the course of computation.

Invalid Operand—This exception occurs if one of the input is a NaN.

Exponent Overflow—This occurs if the result of a computation is such that its rounded result is finite and not an invalid result but its exponent is too large to represent in the floating point format, i.e., exponent has a biased value of 255 or more.

Exponent Underflow—This occurs if the result of a computation is such that its exponent is 0 or less.

Divide-by-zero—This exception occurs if the FLP divide routine is called with F2 being 0.

The package signals exceptions in 2 ways. First, a word at address ERRFLG is maintained that registers the history of the exception conditions. Bits 0–3 of this word are used for the same.

Bit 0 – Exponent overflow detect

Bit 1 – Exponent Underflow detect ERRFLG

Bit 2 – Illegal Operand detect

Bit 3 – Divide–by–0 detect

ERRFLG

*	*	*	*	DBZ	IOP	EUF	EOV

This bits are never cleared by the FLP package, and can be examined by the user software to determine the exception conditions that occurred during the course of a computation. If it is the responsibility of the user software to initialize this word before calling any of the floating point routines.

The second method that the package uses to signal exceptions is to call a user floating point exception handler whenever such conditions occurs. The corresponding exception bit in ERRFLG is set before calling the handler. The starting address of the handler should be defined by the symbol FPTRAP.

Unpacked FLP Format

The IEEE standard FLP format described earlier is very cumbersome to deal with during computation. This is primarily because of splitting of the mantissa between the 2 words. The subroutine in the package unpack the input IEEE FLP numbers into an internal representation, do the computations using this representation, and finally pack the result into the IEEE format before returning to the calling program. The unpacking is done by the subroutine FUNPAK and the packing by the subroutine FPAK. The unpacked format consists of 3 words and is organized as follows:

Unpacked FLP

Fn-Exponent (8-bit biased)	Fn-Sign (extended to 8-bits)
MS 16-bits of Mantissa (implicit 1 is present as MSB)	
LS 8-bits of Mantissa	Eight 0's

Since all computations are carried out in this format, note that the result is actually known to 32 bits. This 32-bit mantissa is rounded to 24 bits before packed to the IEEE format.

Algorithms

All the arithmetic algorithms first check for easy cases when either F1 or F2 is zero or a NaN. The result in these cases is immediately available. The description of the algorithms below is for those cases when neither F1 or F2 is 0 or a NaN. Also, in order to keep the algorithm description simple, the check for underflow/overflow at the various stages is not shown. The documentation in the program, the flowcharts given below, and the theory as described in the references should allow these programs to be easily maintained.

FPADD AND FPSUB

Before a floating point add/subtract instruction is executed, the 2 operands in normalized form

The processing steps are as follows:

1. Compare the 2 exponents.
2. Align the mantissas by equalizing their exponents.
3. Compute result sign as the XOR of the signs of the 2 numbers.
4. Add/Subtract the mantissas.

   For subtract, FP2 is complemented and added with FP1,
   i.e., FSUB = FP1 + (–FP2).
5. Normalize the resulting sum/difference.
6. Pack the exponent, sign and mantissa in IEEE format and return.

FMULT = FP1 * FP2

Floating-point multiplication is accomplished by multiplying the mantissas of the 2 operands and adding their corresponding exponents. Exponent overflow or underflow may occur when true addition is performed on 2 exponents of the same sign.

The processing steps are as follows:

1. Add the 2 exponents and subtract 7FH (IEE bias of 127_{10}) to yield the result exponent.

   Result Exponent = FP1_EXP + FP2_EXP – 127
2. XOR the sign bits to get the result sign.

   Result Sign = FP1_SIGN XOR FP2_SIGN
3. Compute FP1_HI x FP2_HI = C1_HI.C1_LO.
4. Compute FP1_HI x FP2_LO = C0_HI.C0_LO.
5. Add C0_HI + C1_LO = C2_LO.
   If more than 16-bits, then C1_HI += 1.
6. Compute FP1_LO x FP2_HI = C3_HI.C3_LO.
7. Add C3_HI + C2_LO = C4_LO.
   If more than 16-bits, then C1_HI += 1.
8. Normalize mantissa. If MSB of C1_HI =/= 1,
   then result exponent += 1 else left shift C1_HI.C4_LO.
9. Round C1_HI.C4_LO to get result mantissa.
10. Pack the exponent, sign and mantissa in IEEE format and return.

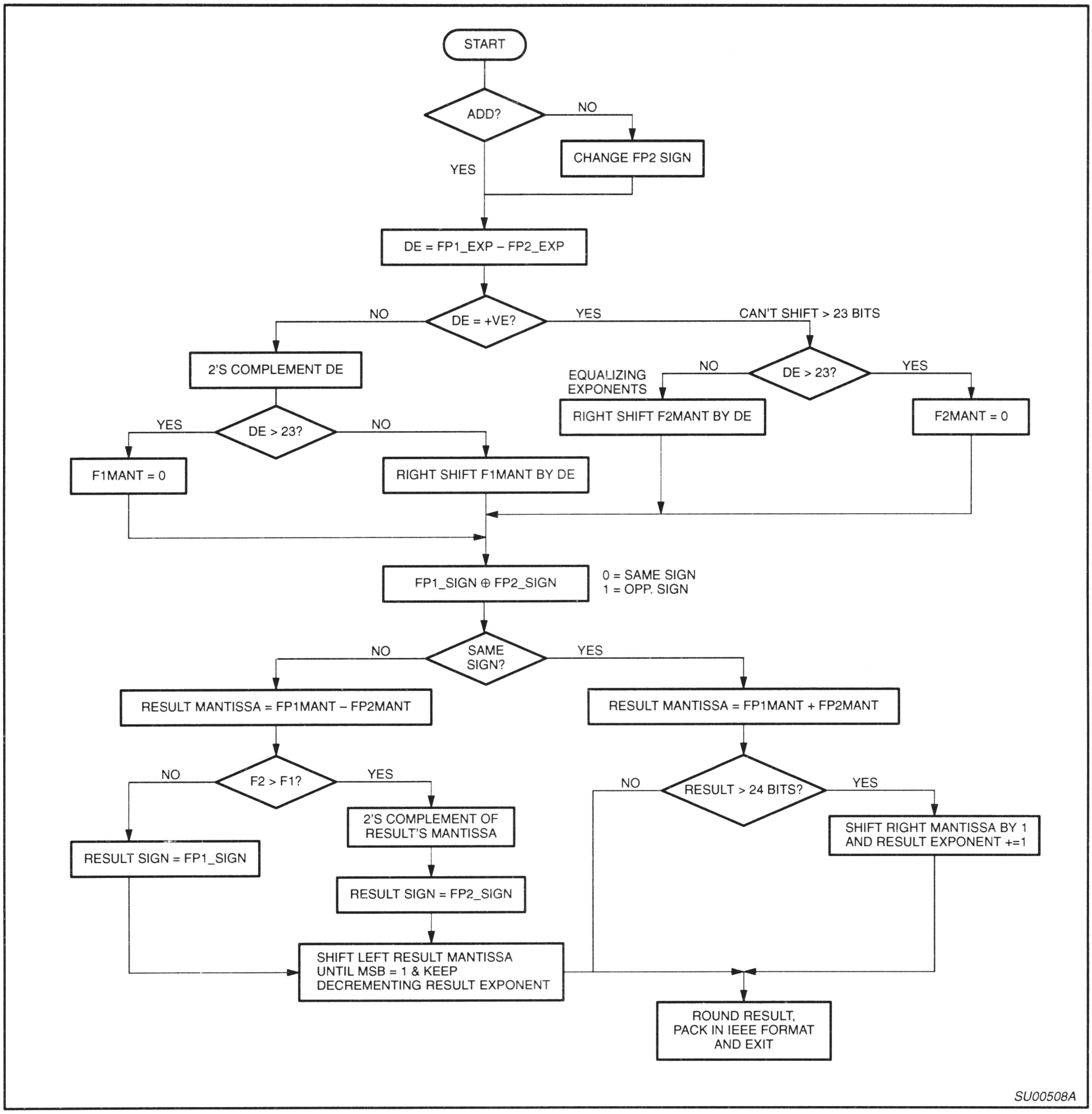
FPDIV = FP1/FP2

The way a floating-point DIVIDE instruction is executed is analogous to that of a Floating Point Multiply, except that mantissa multiplication is replaced by mantissa division and the exponent addition by exponent subtraction. Exponent overflow or underflow may occur when true addition is performed on the 2 exponents of opposite signs. The scheme must avoid the situation of having a divisor which is smaller than dividend mantissa, including the special case of a 0 divisor. With this constraint, the post normalization is unnecessary in FLP division as long as pre-normalization was conducted to avoid quotient overflow.

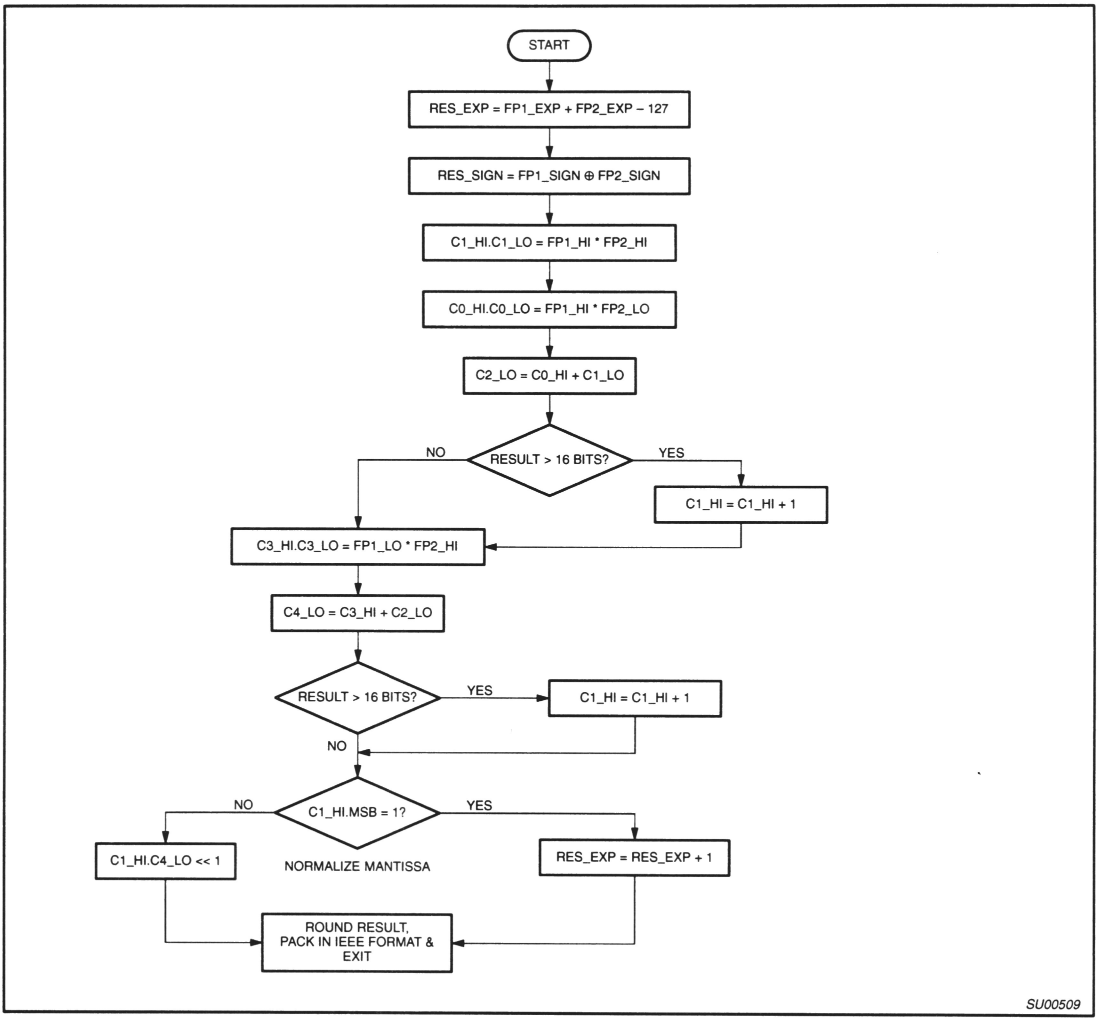
The processing steps are as follows:

1. Compare FP1_HI and FP2_HI.
   If FP2_HI > FP1_HI, then go to step 3, else go to step 2.
2. Shift right FP1 and FP1_EXP += 1.
3. Compute FP1_EXP – FP2_EXP + 127 to get C_EXP.
4. Compute FP1_SIGN XOR FP2_SIGN to get result sign,
   i.e., C_SIGN
5. Compute FP1_HI x FP2_LO = M1_HI.M1_LO.
6. Divide M1_HI.M1_LO / FP2_HI = M2_HI (Quotient)
7. Do a true subtract FP1_LO – M2_HI = M3_LO.
   If result –ve then go to step 8 else FP1_HI –= 1 and go to step 8.
8. Divide FP1_HI.M3_LO / FP2_HI = C1_HI (Quotient) + R1 (remainder)
9. Divide R1.0000 / FP2_HI = C1_LO (Quotient)
10. If MSB of C1_HI = 1, then go to step 11, else shift left
    C1_HI.C1_LO, C_EXP –= 1, and go to step 11.
11. Round C1_HI.C1_LO to get C_HI.C_LO, go to step 12
12. Pack the exponent, sign and mantissa in IEEE format and return.

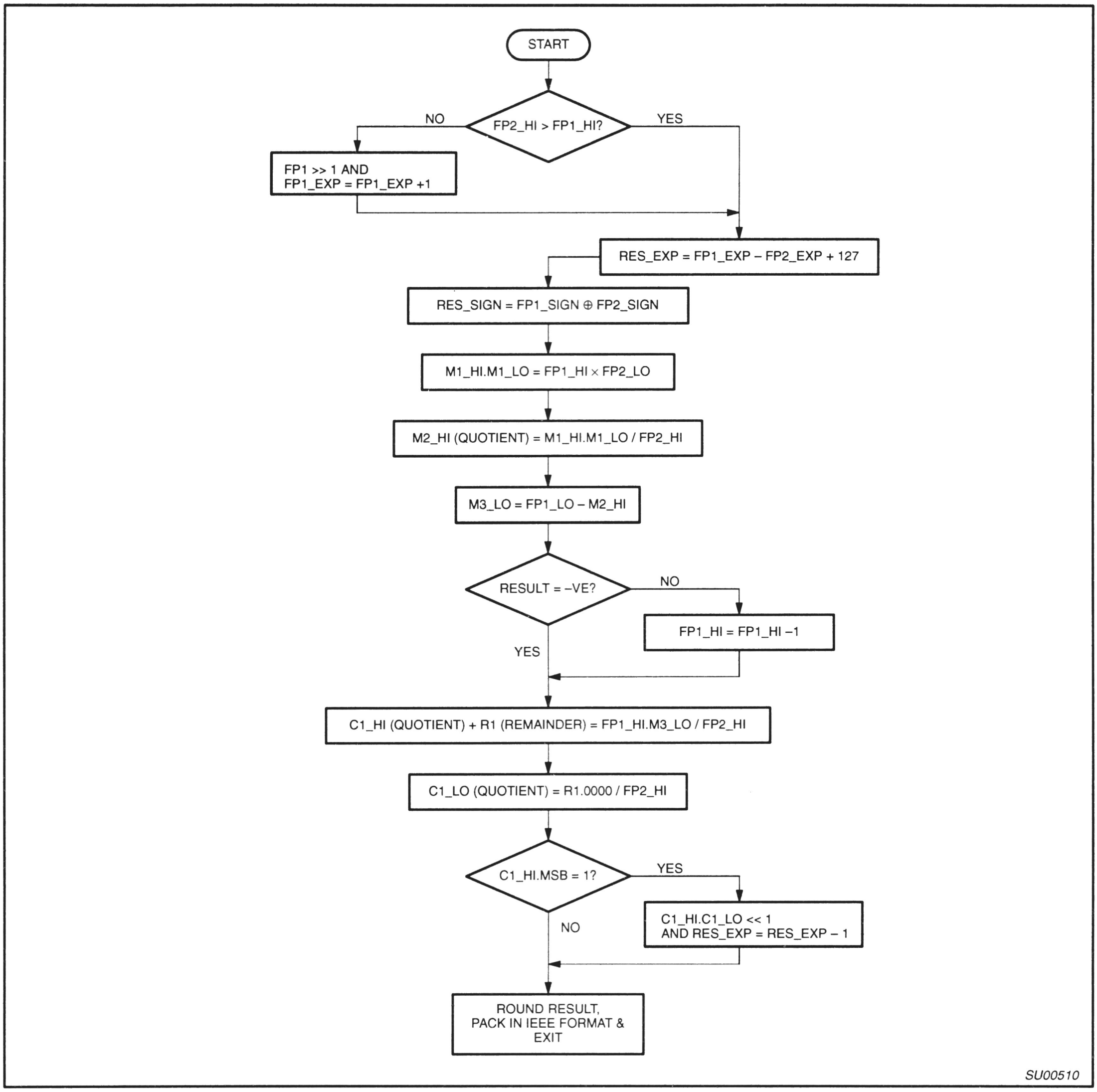
FPADD AND FPSUB


START
ADD?
NO
CHANGE FP2 SIGN
YES
DE = FP1_EXP – FP2_EXP
NO
DE = +VE?
YES
CAN'T SHIFT > 23 BITS
2'S COMPLEMENT DE
EQUALIZING EXPONENTS
NO
DE > 23?
YES
RIGHT SHIFT F2MANT BY DE
F2MANT = 0
YES
DE > 23?
NO
F1MANT = 0
RIGHT SHIFT F1MANT BY DE
FP1_SIGN ⊕ FP2_SIGN
0 = SAME SIGN
1 = OPP. SIGN
NO
SAME SIGN?
YES
RESULT MANTISSA = FP1MANT – FP2MANT
RESULT MANTISSA = FP1MANT + FP2MANT
NO
F2 > F1?
YES
NO
RESULT > 24 BITS?
YES
2'S COMPLEMENT OF RESULT'S MANTISSA
SHIFT RIGHT MANTISSA BY 1 AND RESULT EXPONENT +=1
RESULT SIGN = FP1_SIGN
RESULT SIGN = FP2_SIGN
SHIFT LEFT RESULT MANTISSA UNTIL MSB = 1 & KEEP DECREMENTING RESULT EXPONENT
ROUND RESULT, PACK IN IEEE FORMAT AND EXIT
SU00508A

FPMULT


START
RES_EXP = FP1_EXP + FP2_EXP – 127
RES_SIGN = FP1_SIGN ⊕ FP2_SIGN
C1_HI.C1_LO = FP1_HI * FP2_HI
C0_HI.C0_LO = FP1_HI * FP2_LO
C2_LO = C0_HI + C1_LO
RESULT > 16 BITS?
NO
YES
C1_HI = C1_HI + 1
C3_HI.C3_LO = FP1_LO * FP2_HI
C4_LO = C3_HI + C2_LO
RESULT > 16 BITS?
YES
C1_HI = C1_HI + 1
NO
C1_HI.MSB = 1?
NO
YES
C1_HI.C4_LO << 1
NORMALIZE MANTISSA
RES_EXP = RES_EXP + 1
ROUND RESULT, PACK IN IEEE FORMAT & EXIT
SU00509

FPDIV


START
FP2_HI > FP1_HI?
NO
YES
FP1 >> 1 AND
FP1_EXP = FP1_EXP +1
RES_EXP = FP1_EXP – FP2_EXP + 127
RES_SIGN = FP1_SIGN ⊕ FP2_SIGN
M1_HI.M1_LO = FP1_HI × FP2_LO
M2_HI (QUOTIENT) = M1_HI.M1_LO / FP2_HI
M3_LO = FP1_LO – M2_HI
RESULT = –VE?
NO
FP1_HI = FP1_HI –1
YES
C1_HI (QUOTIENT) + R1 (REMAINDER) = FP1_HI.M3_LO / FP2_HI
C1_LO (QUOTIENT) = R1.0000 / FP2_HI
C1_HI.MSB = 1?
YES
C1_HI.C1_LO << 1
AND RES_EXP = RES_EXP – 1
NO
ROUND RESULT,
PACK IN IEEE FORMAT &
EXIT
SU00510

APPENDIX A

Conversion of Floating Point Numbers

In general IEEE FP = +– mantissa X 2 $^{\text{exponent}}$

Format = Sign bit (1). Biased Exponent bits (8). Normalized Mantissa bits (23), e.g., convert 1.0 to a 32–bit IEEE Floating Point:

1 = 1.0 X 2^0;
Biased Exponent = 0 + 127 = 127 = 7F Hex
The 24-bit normalized mantissa = 100000000000000000000000;
Sign = positive = 0;

So, IEEE 1.0 = 0 0111 1111 00000000000000000000000

which is 3f80 0000 hex & so on.

Some Floating Point Numbers are given below for user reference:

–1.0 = BF80 0000;
+1.0 = 3F80 0000

–0.25 = BE80 0000;
+0.25 = 3E80 0000

–0.50 = BF00 0000;
+0.50 = 3F00 0000

6.250 = 40C8 0000;
1.625 = 3FD0 0000;

12.0 = 4140 0000

References

1. IEEE Draft 8.0 on *A Proposed Standard for Binary Floating-point Arithmetic, 1981*
2. K.Hwang, *Computer Arithmetic, John–Wiley and Sons, 1979*
3. *Microprocessor System Design Concepts, Nikitas A. Alexandridis, Computer Sc.Press, 1984*
4. *Microprocessors and Digital Systems, Douglas V. Hall, McGraw-Hill, 1980*

APPENDIX B

```
#include <stdio.h>
#include "fpp.h"




void main(void)

{

unsigned int fprtn;
unsigned short j;
FILE *fp;


start:
       while(1)        // clear RAM
 {

                  for(j=0; j<6;j++)
                  {
                      tmp1[j] = 0;
                      tmp2[j] = 0;
                  }


                  // Menu

                  printf("\n\n\n\n");
                  printf("             XA FLOATING POINT ARITHMETIC FUNCTION MENU\n");
                  printf("            -------------------------------------------\n");
                  printf("\n");
                  printf("              A .............Floating Point Add\n");
                  printf("              B .............Floating Point Subtract\n");
                  printf("              C .............Floating Point Multiply\n");
                  printf("              D .............Floating Point Divide\n");
                  printf("              H .............Help File\n");
                  printf("              J .............Error Register Status\n");
                  printf("              Q .............Exit Menu\n\n\n\n\n");
                  printf("              ==>");


                  fprtn = getche();   /* wait for user i/p */

                  getch();   // wait for <CR>
                  printf("\n\n");


                  /* Select Floating Point Routine */

                  switch(fprtn)
                  {
                      case 'A' :
                      case 'a' :
                                    printf("Floating Point Addition in Progrss....\n\n\n");
                                    getnum();
                                    fpadd();
                                    break;
```

```
                        case 'B' :
                        case 'b' :
                                       printf("Floating Point Subtraction in Progrss....\n\n\n");
                                       printf("\n\n\n\n");
                                       getnum();
                                       fpsub();
                                       break;

                        case 'C':
                        case 'c':
                                       printf("Floating Point Multiplication in Progress....\n\n\n");
                                       printf("\n\n\n\n");
                                       getnum();
                                       fpmult();
                                       break;



                        case 'D':
                        case 'd':
                                       printf("Floating Point Divide in Progress....\n\n\n");
                                       printf("\n\n\n\n");
                                       getnum();
                                       fpdiv();
                                       break;

                        case 'H':
                        case 'h':
                                       if( (fp = fopen("help","r")) == NULL)
                                       printf("can't open flpdat for read\n");

                                       fcopy(fp,stdout);
                                       fclose(fp);
                                       printf("Hit any key to continue ...");
                                       getch();
                                       goto start;


                        case 'J':
                        case 'j':
                                       show_err_reg();
                                       printf("Hit any key to continue ...");
                                       getch();
                                       goto start;
                        case 'Q':
                        case 'q':
                                       printf("******Hit Ctrl+C to exit ....!!!!*******\n");
                                       getch();


                        default:
                                       printf("   *****************************************************\n");
                                       printf("   *  UNKNOWN COMMAND, GOODBYE!!                       *\n");
                                       printf("   *****************************************************\n");
                                       goto start;

  }

 }

}
```

```
/* Exception Handling Routines */

void divbyz (void)

{
                ERRFLG |= 8;            /* set the DIVBY0 bit (#3) */
                fp_lo = 0;                          /* Return  NaN */
                fp_hi = NANH;
                fptrap();               /*  exception handler */
                return;
}


/* Illegal Operand - one of the numbers is a INVALID. */

void fnan(void)

{

                ERRFLG |= 4;            /* set the NAN operand bit */
                fp_lo = NANL;
                fp_hi = NANH;           /* return NAN in fp_lo and fp_hi */
                fptrap();
                return;
}



void undflw(void)               /* exponent underflow */

{
                ERRFLG |= 2;            /* set the exponent underflow bit */
                fp_lo = 0;              /* clear FP */
                fp_hi = 0;
                fptrap();
                return;
}

void ovflw(void)                /* exponent overflow */

{
                ERRFLG |= 1;            /* set the exponent overflow bit */
                fp_lo = 0;
                fp_hi = NANH;
                fptrap();
                return;
}


/* Subroutine to check if a single precision FLP # stored in the
   IEEE flp format in registers fp_lo and fp_hi is 'INVALID'
   Returns 0 if number != INVALID and
   Returns 1 if Number == INVALID
*/

unsigned int  fnanchk()

/* Subroutine to check if a SP floating point number is a NaN
   Returns 1, if YES, and 0 if NOT */

{
                long tmp;

                tmp = fp_hi;
                tmp = tmp >> 1;         /* Shift left fp_hi by 1 */
                if(tmp > 0xfeff)        /* feff + 1 = ff00 */
                return 1;               /* biased exp >= 255 & f =/= 0 */
                return                  /* OK */
}
/* Subroutine to check if a single precision FLP # stored in the
   IEEE flp format in registers fp_lo and fp_hi is 'ZERO'
   Returns 0 if number != and
   Returns 1 if Number == 0
   Note : fp "0" = 1.0 x 2 e -127  i.e if biased exp = 0, fp = 0
*/

char zchk(void)

{
                unsigned long tmp;
                tmp = fp_hi;

                tmp = tmp << 1;         /* Shift left fp_hi by 1 */

                if(tmp > 0x00ff)
                return 0;
                return 1;
}


/* subroutine to unpack a SP IEEE formatted FP # and held in regs.fp_hi
   and fp_lo.  The unpacked format occupies 3 words & is organized as
   follows:

   WORD2 :  eeeeeeee ssssssss     -> biased-exp.sign
   WORD1 :  1mmmmmmm mmmmmmmm     -> 16 MSB of Mantissa (m23 : m16)
   WORD0 :  mmmmmmmm 00000000     -> 8 LSB of Mantissa.zeros

   e7:0 - 8-bit exponent in excess-127 format
   s7:0 - sign bit -> 0x00 = +ve, 0xff = -ve ;
   m23:0 - normalized mantissa i.e 1.0000...
*/

char* funpak(fparray)                 // returns ptr. to a character array to fpadd
int *fparray;                         //pointer to the flp. array passed by fpadd

{

                char  sign;
                unsigned short i= 0,j = 0;

                     if(k=2) k=0;
                     else
                     k ++;
                     flpar[i] = 0;   // clear lo-byte of fp0



                flpar[++i] = fparray[j] & 0x00ff; /* fp0_hi = m0:m7 */
                flpar[++i] = (fparray[j++] & 0xff00) >> 8; /* fp1_lo = m8:m15 */


                flpar[++i] = fparray[j] & 0x007f; /* m16:m22 */


                flpar[i] |= 0x80;                  /* set bit7 -> normlz. bit */


                sign = (fparray[j] & 0x8000) >> 15; /* check sign bit */
                                                     /* fp2_lo = 8 sign bits */

                  if (sign)                          /*  -ve number ? */

                    flpar[++i] = 0xff;
                  else                               /* yes */
                    flpar[++i] = 0x00;               /* no */

      flpar[++i] = (fparray[j] & 0x7f80) >> 7;       /**/
                return (flpar);                      /* return the ptr to the array */
}
```

```
/* subroutine to pack a SP held in 3 words flpar0:3 into IEEE format.
   The packed format is stored in fp_hi & fp_lo as follows:
   fp_hi :  seeeeee  emmmmmmm     -> sign.biased-exp.7 MSBs of mantissa
   fp_lo :  mmmmmmmm mmmmmmmm     -> 16 LSBs of Mantissa

                Result Stored in tmp2[i]; i=0:5
                tmp2[0] = 0x00; tmp[1] = m0:m7; tmp[2] = m8:m15;
                tmp[3] = 1.m16:m22; tmp[4] = sign7:0; tmp[5] = exp7:0
*/


void fpak(void)

{
                int i=1;
                unsigned int sign,tmpc;
                long ltmp1,ltmp2;


                fp_lo =  tmp2[i++];                 /* get m0:m7 */
                tmpc = tmp2[i++];
                fp_lo = fp_lo | (tmpc << 8);        /* get m8:m15 */


                fp_hi = tmp2[i++];                  /* get 1.m22:m16 */
                ltmp1 = (fp_hi & 0x007f);
                ltmp1 = ltmp1 << 16;                /* mask & shift */


                sign = tmp2[i++];                   /* save sign-bit(s) */
                ltmp2 = tmp2[i];                    /* exponent */
                ltmp2 = ltmp2 << 23;
                ltmp2 = ltmp2 | ltmp1;

                if(sign)
                ltmp2 = ltmp2 | 0x80000000;

                printf("\n\nThe IEEE packed result is:");
                printf("%lx\n",ltmp2);
}


/* This routine rounds up the 32-bit mantissa resulted from FP operations
   to a 24-bit number */

void fround(void)
{

    unsigned int i=3, ctmp;
    long tmpl;

                tmpl = tmp2[i--];    /* 1.m22:16 */
                tmpl = tmpl << 8;
                ctmp = tmp2[i--];    /* m15:8 */
                tmpl = tmpl|ctmp;
                tmpl = tmpl << 8;
                ctmp = tmp2[i--];    /* m7:0 */
                ctmp = tmp2[i];

                    if(ctmp & 0x80)   /* if bit 7 i not set in the LSB */

                                 tmpl += 0x0100;  /* Increase next byte by 1 */

                                 ctmp = tmp2[5];

                                   if(tmpl & 0x1000000)    /* carry out of MSB? */
                                   {
                                                                  tmpl = tmpl >> 1;
                                                                  ctmp++;                  /* exp = exp+1 */

                                                                    if(ctmp > 255)

                                                                                 tmp2[5] = 0xff;
                                   }


                                                  tmp2[1] = (tmpl & 0x000000ff);
                                                  tmp2[2] = (tmpl & 0x0000ff00) >> 8;
                                                  tmp2[3] = (tmpl & 0x00ff0000) >> 16;
}


/* User supplied FLP Trap Routine */


void fptrap(void)

{
                printf("\n\nException Occurance !!!\n\n");

                printf("Error Flag Register = %x",ERRFLG);
                /* User exception handler

                ....
                ....
                ....
                ....
                ....
                ....
                ....
                */

}


void show_err_reg(void)

{
    printf("\n The Error Flag Register Bit Map is as follows :");
    printf("\n --------------------------------------\n\n");
    printf("       ---------------------------------------\n");
    printf("      | - | - | - | - | DBZ | IOP | EUF | EOV |\n");
    printf("       ---------------------------------------\n\n\n");
    printf(" where DBZ = Divide by Zero exception\n");
    printf("       IOP = NaN or Invlaid operand\n");
    printf("       EUF = Exponent Underflow\n");
    printf(" and   EOV = Exponent Overflow\n\n\n");


    printf("The status of the register after the operation is :");
    printf("0x%x\n\n",ERRFLG);
 }
/* FPADD - Floating Point Add
   It is assumed two  floating point numbers F1 & F2 are in IEEE format
   Format :
   Fn = fpnh (s.e7:0.m22:16) + fpnl (m15:0)  -> In registers
*/


int fpadd()

{               // 1

long dmant, flpm1, flpm2, lnfp, tlong;
int  *flpn, temp1, temp2;
char dexp, i=0,j=0,k=5,t=0, exp1, exp2;

                exp1 = tmp1[k];   /* get exponent of 1st */

                exp2 = tmp2[k];   /* get exponent of 2nd */

                dexp = (exp1 - exp2);           /* difference in exponent */

                printf("\n\n");
                printf ("DEXP =  %x\n", dexp);
```

```
/* CASE 1: */
                                 if(dexp > 0 && dexp < 23)        /* f1exp > f2exp */
  {  //2


                                 tmp = dexp;

                  /* if exp > 23, can't shift mantissa
                                                                 more than 23-bits */


                                 while(tmp--)
                                 {
                                   for(i=0; i<=3; i++)
                                   {
                                   tmp2[i] = tmp2[i] >> 1;
                                   }
                                  }



                                  i = 4;
                                  temp1 = tmp1[i] & 0x00ff;  /* get the fp1 sign byte */
                                  printf("\nFP1 SIGN BITS = %x", temp1);

                                  temp2 = tmp2[i] & 0x00ff;  /* get the fp2 sign byte */
                                  printf("\nFP2 SIGN BITS = %x", temp2);



loop_here:
                                  i = 4;
                                  rsign = (tmp1[i] ^ tmp2[i]);  /* ex-or sign bits */

                                  if(rsign != 0)                   /* if different sign */

                  { //44
                                  printf("\nFPs ARE OF DIFFERENT SIGNS!!\n");

                                  flpm1 = getmant(0);
                                  flpm2 = getmant(1);

                                  dmant = flpm1 - flpm2;

                                  if (flpm1 > flpm2)               /* F1man > F2man */

                 {  //22

                                  printf("\n");
                                  printf("FP1 MANTISSA GREATER THAN FP2 MNATISSA\n");
                                  printf("\n\n");

                                  tmp2[4] = tmp1[4];              /* result sign = sign of F1 */
                                  rsign = tmp1[4];                /* stored in FP2 sign byte */


                  /* Shift left mantissa till MSB = 1 */

                            for(k=0; k <= 23  && ((dmant & 0x00800000) == 0); k++)
                                 {
                                                              dmant = dmant << 1;
                                                              exp1 = exp1 - 1;
                                 }


                            /* save result in tmp2[i] array */
                                            tmp2[5] = exp1;
                                            tmp2[4] = rsign;
                                            tmp2[3] = (dmant & 0x00ff0000) >> 16;
                                            tmp2[2] = (dmant & 0x0000ff00) >> 8;
                                            tmp2[1] = (dmant & 0x000000ff);
```

```
              printf("RESULT EXPONENT : %x\n", exp1);
              printf("RESULT MANTISSA (NORMLZD) = %lx\n", dmant);
              printf("RESULT SIGN = %x\n", rsign & 0x1);

                              fround();   /* round the result */
                                    fpak();                       /* Pak & leave */

                                    printf("Hit any key to continue ...");
                                    getch();
                                    return 0;

        } //22
              else if (flpm1<flpm2)              /* F2man > F1man */

        { //23
                              printf(" FP2 MANTISSA GREATER THAN FP1 MANTISSA \n");


                              dmant = ~dmant; /* 2'S COMPLEMENT */
                              dmant++;
                              exp2 = tmp2[5];     /* res.exp = F2.exp */
                              rsign = tmp2[4];
                              tlong = dmant;

              while(!(tlong & 0x800000))
                              { dmant = dmant << 1;
                                     tlong = dmant;
                                     exp2--;
                              }

                             dmant = dmant & 0xffffff;



                           /* save result in tmp2[i] array */
                             tmp2[5] = rexp;
                             tmp2[4] = rsign;
                             tmp2[3] = (dmant & 0x00ff0000) >> 16;
                             tmp2[2] = (dmant & 0x0000ff00) >> 8;
                             tmp2[1] = (dmant & 0x000000ff);

                           printf("\n\n");
                           printf("THE RESULT MANTISSA (NORMLZD) IS = %lx\n",dmant);
                           printf("THE RESULT EXPONENT IS = %x\n", exp2);
                           printf("THE RESULT SIGN IS = %x\n", rsign & 0x1);

                              fround();
                              fpak();
                              printf("Hit any key to continue ...");
                              getch();
                                                  return(0);
        } //23



  } // 44

        else if(rsign == 0) // same sign, so ex-OR is 0
```

```
     { //55
                         printf("\n");
                         printf("FPs ARE OF SAME SIGN!!\n");

                                            flpm1 = getmant(0);
                                            flpm2 = getmant(1);
                                            dmant = flpm1 + flpm2;
                                            rsign = tmp1[4];
                                            rexp = exp1;

                                            tlong = dmant;
                                            tlong = tlong & 0x01000000;// check if carry set


                                            if(tlong)
                                         {
                                         dmant = dmant >> 1;
                                         rexp = rexp + 1;
                                         }

                                         /* save result in tmp2[i] array */
                                            tmp2[5] = rexp;
                                            tmp2[4] = rsign;
                                            tmp2[3] = (dmant & 0x00ff0000) >> 16;
                                            tmp2[2] = (dmant & 0x0000ff00) >> 8;
                                            tmp2[1] = (dmant & 0x0000ff);


                                         printf("\n\n");
                                         printf("THE RESULT MANTISSA(NORMLZD) IS = %lx\n",dmant);
                                         printf("THE RESULT EXPONENT IS = %x\n", rexp);
                                         printf("THE RESULT SIGN IS = %x\n", rsign & 1);


                                           fround();
                                           fpak();      /* pack in IEEE format */

                         printf("Hit any key to continue ...");
                                         getch();

                                            return(0);
     } //55


} // 2



                         else if (dexp > 23)

{ //99
                                         printf("DIFFERENCE IN EXPONENT IS  GREATER THAN 23\n");

                                         for(i = 0; i<= 3; i++)
                                         tmp2[i] = 0;
                                         goto loop_here;
} //99



     else if (dexp < 0)
```

```
{//6
               printf("DIFFERENCE IN EXPONENT IS NEGETIVE\n");
               printf("FP2 EXPONENT GREATER THAN  FP1 EXPONENT\n");

               tmp = ~dexp ;     /* 2's complement */
               tmp++;

       printf("2's COMPLEMENT OF DEXP = %d\n", tmp);


                                 if(tmp < 23)
     {
                                 while(tmp--)
                                 {
                                   for(i = 0; i<= 3; i++)
                                   {
                                   tmp1[i] = tmp1[i] >> 1;
                                   }
                 }
                    goto loop_here;
     }

                                   else if(tmp > 23)               /* dexp > 23 */
    {
                                   for(i=0; i<= 3; i++)
                                   tmp1[i] = 0;                    /* F1mant = 0 */
                                   goto loop_here;
    }

  } //6
                    else if(dexp == 0)                             /* shift done */
                      printf("FPs GOT SAME EXPONENT!!\n");
                    goto loop_here;

} // 1


/* Get the mantissa for both FP in 24-bit format */


long getmant(val)
 unsigned short val;

{
    long lnfp, flpm;
    unsigned short i;
                                   i= 1;
                                   lnfp = 0;
                                   flpm = 0;

    if (!val)
                    {

                                   lnfp = tmp1[i];
                                   flpm |= (lnfp & 0x000000ff);

                                   lnfp = 0;
                                   i++;
                                   lnfp = tmp1[i];
                                   lnfp = lnfp << 8;
                                   flpm |= lnfp & 0x0000ff00;


                                   lnfp = 0;
                                   i++;
                                   lnfp = tmp1[i];
                                   lnfp = (lnfp << 16);
                                   flpm |= (lnfp & 0x00ff0000);


                                   flpm = flpm & 0x00ffffff;

                                   printf("\n FP1 MANTISSA = %lx\n", flpm);

                    }
                    else
                       {
                                    i=1;
                                    flpm = 0;
                                    lnfp = 0;
                                    lnfp = tmp2[i++];
                                    flpm |= (lnfp & 0x000000ff);

                                    lnfp = 0;
                                    lnfp = tmp2[i++];
                                    lnfp = (lnfp << 8);
                                    flpm |= (lnfp & 0x0000ff00);

                                    lnfp = 0;
                                    lnfp = tmp2[i];
                                    lnfp = (lnfp << 16);
                                    flpm |= (lnfp & 0x00ff0000);

                                    flpm = (flpm & 0x00ffffff);

                                    printf(" FP2 MANTISSA = %lx\n", flpm);

                       }
     return (flpm);
}


int fpsub()
{
   unsigned char fp2_sign;

   fp2_sign = tmp2[4];
   fp2_sign = ~fp2_sign;

   tmp2[4] = fp2_sign;
   fpadd();
}


void getnum()

{
unsigned int fp[3], px;
unsigned short i,j,k;

printf("Type the lo-word of FP#1 in IEEE format :");
                 scanf("%x",&px);
                 i= 0;
                 fp[i] = *&px;

                 printf("Type the hi-word of FP#1 in IEEE format :");
                 scanf("%x",&px);
                 fp[++i] = *&px;


                 funpak(fp);  // pass the array ptr. to unpack routine

                 for(k=0,j=0; k<=5; j++,k++)
                 {
                 tmp1[k] = flpar[j];
                 }
                 printf("\n");


                 i=0;
                 printf("\n\n");
                 printf("Type the lo-word of FP#2 in IEEE format :");
                 scanf("%x",&px);
                 fp[i] = *&px;
```

```
                  printf("Type the hi-word of FP#2 in IEEE format :");
                  scanf("%x",&px);
                  fp[++i] = *&px;
                  funpak(fp);

                  for(k=0,j=0; k<=5; j++,k++)
                  {
                  tmp2[k] = flpar[j];
                  }

}

void fpmult()
{

    int fp1_exp,fp2_exp, res_sign;
    unsigned int fp1_hi, fp1_lo, fp2_hi, fp2_lo, tmp;
    unsigned int c0_hi,c0_lo,c1_hi, c1_lo, c2_lo,c3_hi, c3_lo, c4_lo;
    int res_exp;
    long ltmp;


    fp1_exp = tmp1[5];
    fp2_exp = tmp2[5];

                  res_exp = (long)(fp1_exp + fp2_exp -127); // result exponent


                            if(res_exp < 0)      // underflow
                            undflw();


                            if (res_exp > 255)
                            ovflw();


                  res_sign = tmp1[4] ^ tmp2[4];  // XOR = result sign


                   tmp = tmp1[3] << 8;    /* 1.m22:8 = FP_HI*/
                   tmp = tmp & 0xff00;
                   tmp |= tmp1[2];

                   fp1_hi = tmp;
                   fp_hi = fp1_hi;

                   tmp = fnanchk();

                            if(tmp)
                            fnan();         // F1 is a NaN


                   tmp = tmp2[3] << 8;    /* 1.m22:8 = FP_LO */
                   tmp = tmp & 0xff00;
                   tmp |= tmp2[2];

                   fp2_hi = tmp;
                   fp_hi = fp2_hi;

                            tmp = fnanchk();

                            if(tmp)
                            fnan();         // F2 is a NaN


                   tmp = zchk();  // Check for F2 = 0

                            if(tmp)  {
                            printf("\nResult is = 0 as Multiplier is 0\n");   // F2 = 0
                            goto endprog; }

                   fp_hi = fp1_hi;
```

```
tmp = zchk();  // Check for F1 = 0

                if(tmp) {
                printf("\nResult is = 0 as Multiplicand is 0\n");    // F1 = 0
                goto endprog;  }


tmp = tmp1[1] << 8;    /* m7:0.0000 */
tmp = tmp & 0xff00;
tmp |= tmp1[0];

fp1_lo = tmp;

tmp = tmp2[1] << 8;
tmp = tmp & 0xff00;
tmp |= tmp2[0];

fp2_lo = tmp;

ltmp =  (long)fp1_hi * (long)fp2_hi;  /* FP1_HI * FP2_HI */


c1_hi = (ltmp &  0xffff0000) >> 16;
c1_lo = ltmp & 0x0000ffff;

ltmp = (long)fp1_hi * (long)fp2_lo;
c0_hi = (ltmp  & 0xffff0000) >> 16;
c0_lo = ltmp &0x0000ffff;

ltmp  = c0_hi + c1_lo;

              if(ltmp & 0x10000)
                  c1_hi++;

c2_lo = ltmp & 0xffff;


ltmp = (long)fp1_lo * (long)fp2_hi;

c3_hi = (ltmp & 0xffff0000) >> 16;
c3_lo = ltmp & 0x0000ffff;

ltmp = c3_hi + c2_lo;

             if(ltmp & 0x10000)
                 c1_hi++;


c4_lo = ltmp & 0xffff;


ltmp = c1_hi;
ltmp = ltmp <<8;
ltmp = (ltmp & 0xffffff00) | c4_lo;

if(!(c1_hi & 0x8000))
ltmp = ltmp << 1;

else
res_exp++;

if(res_exp > 254)
ovflw();


/* save result in tmp2[i] array */
                 tmp2[5] = res_exp;
                 tmp2[4] = res_sign;
                 tmp2[3] = (ltmp & 0x00ff0000) >> 16;
                 tmp2[2] = (ltmp & 0x0000ff00) >> 8;
                 tmp2[1] = (ltmp & 0x000000ff);
```

```
                    printf("\n\n");
                    printf("RESULT EXPONENT : %x\n", res_exp);
                    printf("RESULT MANTISSA (NORMLZD) = %lx\n", ltmp);
                    printf("RESULT SIGN = %x\n", res_sign & 1);

                    fround();


                  /* final check of exponent */

                    tmp = tmp2[5];

                                  if(!tmp)
                                  undflw();  /* exponent underflow */

                                  if (tmp > 254)
                                  ovflw();

                                  fpak();
endprog:
                    printf("Hit any key to continue ...");
                                       getch();
}


void fpdiv(void)

{
    unsigned int tmp, fp1_hi, fp2_hi, fp1_lo, fp2_lo, c1_hi, c1_lo;
    unsigned char i, c1_exp, c1_sign, fp1_sign, fp2_sign;
    long ltmp, rem, lfpm1, lfpm2, tmplng;
    unsigned int m1_hi, m1_lo, m2_hi, m2_lo, m3_hi, m3_lo;
    signed int fp1_exp, fp2_exp, dexp;

                    tmp = tmp1[3] << 8;
                    tmp = tmp & 0xff00;
                    tmp |= tmp1[2];

                    fp1_hi = tmp;
                    fp_hi = fp1_hi;
                    tmp = fnanchk();

                                  if(tmp)
                                  fnan();        // F1 is a NaN

                    tmp = tmp2[3] << 8;
                    tmp = tmp & 0xff00;
                    tmp |= tmp2[2];

                    fp2_hi = tmp;
                    fp_hi = fp2_hi;

                    tmp = fnanchk();

                                  if(tmp)
                                  fnan();        // F2 is a NaN


                                  tmp = zchk();  // Check for F2 = 0

                                  if(tmp)  {
                                  divbyz();      // F2 = 0
                                  printf("\nException as a result of Division by 0\n");
                                  goto exit;
                                  }

                                  fp_hi = fp1_hi;
                                  tmp = zchk();  // Check for F1 = 0

                                  if(tmp)        // Result = 0
                                  {
                                  printf("\nResult of Division of a zero Dividend is = 0\n");
                                  goto exit;
                                  }
```

```
                tmp = tmp1[1] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp1[0];

                fp1_lo = tmp;


                tmp = tmp2[1] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp2[0];

                fp2_lo = tmp;


                lfpm1 = fp1_hi;
                lfpm1 = lfpm1 << 16;
                lfpm1 |= fp1_lo;

                lfpm2 = fp2_hi;
                lfpm2 = lfpm2 << 16;
                lfpm2 |= fp2_lo;


/* Exponent bits */
                fp1_exp = tmp1[5];

                fp2_exp = tmp2[5];


/* Sign bits */
                fp1_sign = tmp1[4];
                fp2_sign = tmp2[4];


    /* Ensure that fp2_hi > fp1_hi */


    if(fp1_hi > fp2_hi)           // compare fp1_hi & fp2_hi

                {
                lfpm1 >> 1;
                fp1_exp++;
                }

   /* else := good */


     fp1_hi = (lfpm1 & 0xffff0000) >> 16;
     fp1_lo = lfpm1 & 0x0000ffff;


     dexp = (fp1_exp - fp2_exp);  // difference in exponent (2's compl)


     c1_exp = dexp + 127;
     c1_sign = fp1_sign + fp2_sign;

     ltmp = (long)fp1_hi*(long)fp2_lo;  // m1_hi.m1_lo

     m1_hi = (ltmp & 0xffff0000) >> 15;
     m1_lo = (ltmp & 0x0000ffff);


     m2_hi = ltmp / (long)fp2_hi;   // Quotient
     m3_lo = fp1_lo - m2_hi;

     if( m2_hi > fp1_lo)    // B = 0 i.e C = 1
                 fp1_hi--;
                 ltmp = (unsigned long)fp1_hi;
                 ltmp = ltmp << 16;
                 ltmp = ltmp|m3_lo;
                 tmplng = ltmp;
```

```
                  ltmp = ltmp /(unsigned long)fp2_hi;  // Quotient
                  c1_hi = ltmp & 0x0000ffff;
                  ltmp = tmplng;
                  ltmp =  ltmp % (unsigned long)fp2_hi; // remainder


                  ltmp = ltmp << 16;
                  ltmp = ltmp & 0xffff0000;
                  c1_lo = ltmp/(unsigned long)fp2_hi;


                  if(c1_hi & 0x8000)

                  {
                     c1_hi << 1;
                     c1_lo << 1;
                     c1_exp -= 1;
                  }


                     ltmp = c1_hi;
                     ltmp = ltmp << 16;
                     ltmp = ltmp | c1_lo;


                     /* save result in tmp2[i] array */
                                       tmp2[5] = c1_exp;
                                       tmp2[4] = c1_sign;
                                       tmp2[3] = (ltmp & 0xff000000) >> 24;
                                       tmp2[2] = (ltmp & 0x00ff0000) >> 16;
                                       tmp2[1] = (ltmp & 0x0000ff00) >> 8;
                                       tmp2[0] = ltmp & 0x000000ff;


                     printf("\n\n");
                     printf("RESULT EXPONENT : %x\n", c1_exp);
                     printf("RESULT MANTISSA (NORMLZD) = %lx\n", ltmp);
                     printf("RESULT SIGN = %x\n", c1_sign & 1);


                                       fround();   // round up results in IEEE format

 /* final check of exponent */

                     tmp = tmp2[5];

                                       if(!tmp)
                                       undflw();  /* exponent underflow */

                                       else if (tmp > 0xfe)
                                       ovflw;     /* exponent overflow */


                                         fpak();  // pack in IEEE format
exit:
                     printf("Hit any key to continue ...");
                                           getch();

}

void fcopy(FILE *ifp, FILE *ofp)
{
    int c;
    while ((c=getc(ifp)) != EOF)
                  putc(c,ofp);
}
```

High level language support in XA

AN702

Author: Santanu Roy, Philips Semiconductors, MCO Applications Group, Sunnyvale, California

Introduction

High Level Language (HLL) support is becoming a key feature in modern day microcontroller architecture. The reason is highly visible. It is easier to code a processor in a high-level platform than in conventional assembly because it is portable, i.e., it is not tied to any one machine. Also, the advantage of coding in a high-level language is because it is modular and re-usable which speeds up any code development process considerably.

In recent years, C has been "the language" of choice for all engineers. Thus almost all modern day microcontrollers are designed with C-language support in mind. This article highlights some of the architectural features of Philips XA microcontroller that has been designed to support such languages specifically C.

Supporting HLL

One of the tasks that an architect has to confront is the determination of exactly what instructions should form the functional instruction of a microcontroller to meet high-level language support. An answer to this is to provide an operation code for each functional operation in a high-level programming language. Thus operation codes will exist for +, –, *, /, and so on. Special provision is made for operation on arrays, and all operations that can be applied to data types in a high-level language are directly supported in the architecture. An instruction set ideally should contain only instructions that are used in a HLL, and not implement any non-functional instructions, i.e., instruction that is not expressed as a verb or operator in a high-level language. Thus "LOAD", "STORE", and so on which are not statements made in high-level languages are redundant and only adds to architectural overheads.

An instruction word consists of a single op-code and an operand address for each HLL variable involved in the operation. *Op-codes are symmetric in that they are applicable to any type of addressing and any data type.*

Some general criteria for an ideal architecture could be:

1. Only one instruction should be executed for most common HLL operators.
2. There should be only one memory reference for each referenced operand.
3. There should be explicit addressing only for operands whose location cannot be inferred by recent processing activity, and address should be short.
4. Instructions should be compact, and densely coded.

The XA Architecture

The XA is a register based machine. Hence most variables could be stored in these fast storage registers for high code density and fast execution. However, the beauty of the XA architecture is that, it is optimized for internal memory as well for high throughput and code density, e.g., a register-register ALU operation takes 2 bytes and 3 clocks and the same ALU operation between register-memory (indirectly addressed) is 2 bytes and 4 clocks. So, a large set of variables could be stored in memory with very little loss in performance. Additionally, hooks like "burst mode", etc., are provided to speed up external memory access as well.

Data Types and Sizes

XA directly supports the following basic data types as used in C:

character (char) – signed and unsigned bytes
integer (int) – signed and unsigned words

Constants – Supported as byte/word (char/int) immediate data in the instructions, e.g., ADD R0, #1234 etc. The range is +32,767 to –32,768 for signed and 0 to 65,536 for unsigned word/integer constants, +127 to –128 for signed or 0 to 255 for unsigned bytes/char.

For "short" qualifier, the range is +7 to –8 as used with instructions MOVS and ADDS.

A "long" qualifier to integer is implemented by the compiler by extending (signed/unsigned) the word to the next higher address(+1). In addition to the above,

Bit – This special data type is also supported to access the different bit addressable space in the machine.

Note: All signed data are represented in 2's complement form in the XA.

Type conversion

All operations are performed under natural data sizes, e.g., MULU.b does a 8x8 unsigned multiply of 2 bytes, MULU.w does the same but with 2 word-size operands. So when operands of different types appear in an expression, they are converted to a common type by the compiler, e.g., operation between a *char* (byte) and an *integer* (word) is promoted to *integer-integer*, etc.

Arrays

XA supports addressing byte and word arrays in memory as required by C or any HLL. Offset and auto-increment addressing modes in XA allow easy access and manipulation of array elements. Offsets are signed values of 8 or 16 bits and are used depending on the size of the array.

Static Variables

Static variables unlike automatic provide permanent storage in a function. This means these variables are stored in memory rather than being a part of run-time stack. A wide variety of memory addressing modes are supported in the XA to provide easy access to static variables in memory. In addition to several indirect addressing modes (auto-increment offset) the XA supports direct access to the first 1K of the memory space in each segment. This is ideal for addressing static variables, and has found to generate extremely dense code. A listing of operations to access static variables is given below for reference:

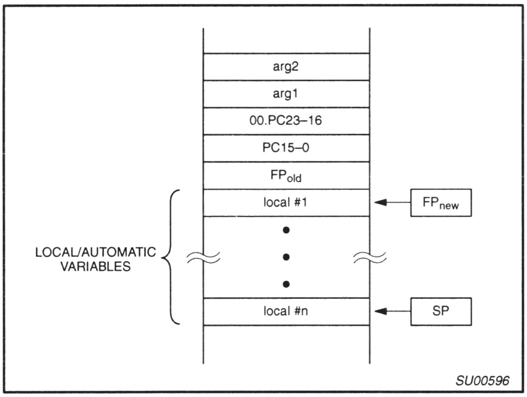
Table 1. Access to Static Variables

ADDRESSING MODES
Rd, direct
direct, Rd
Rd, [Rs+]
[Rs+], Rd
Rd, [Rs+]
Rd, [Rs+Offset8/16]
[Rs+Offset8/16], Rd
direct, direct
[Rd], #immediate
direct, #immediate
[Rd+], #immediate
[Rd+], [Rs+]

Automatic/Dynamic Variables

Within a function, a typical compiler maintains a Frame Pointer (FP), which is used to access function arguments and local automatic variables. To call a function, a compiler pushes arguments onto the stack in reverse order, (the PUSH instruction decrements the SP by 2 each time it is executed, calls the function, then increments the SP by the number of bytes pushed. For instance, to call a function with two one-word arguments, the XA-compiler generates code to do the following:

```
PUSH arg2         ; (SP –=2)
PUSH arg1         ; (SP –=2)
CALL (subroutine) ;
ADDS SP,4         ; (SP +=4)
```



The CALL instruction pushes the current PC onto the stack. Because all stack pushes are 16-bits in XA, any 8-bit function argument is automatically promoted to word.

Upon function entry, the compiler creates new stack and frame pointers by computing:

```
PUSH FP (old)
FP (new) = SP
SP = SP – Framesize;
```

where "Framesize" is the space required for all local automatic variables. If the frame size is odd, the compiler always rounds it up to the next even number. If there are 2 arguments and 2 local variables, then the frame size is 4 and the stack looks like this:

FP+8 second argument
FP+6 first argument
FP+4 return address
FP+2 old FP
FP–0 first local variables
FP–2 second local variable
FP–4 next free stack location (same as SP)

If a function argument is defined to be an 8-bit type, then only the lower 8-bits of the value pushed by the caller are to inside the called function.

Upon function exit, the compiler restores the SP and FP to their original value by executing the following:

```
SP = FP
POP FP
RET
```

The return instruction RET sets the new PC by popping the saved PC off the stack.

Because there are so many registers in XA (unlike 8051), any of them could be assigned to hold the FP. Access to variables in the stack space is easily achieved through the indirect-offset addressing modes (signed 8 or 16) with respect to the stack pointer. In almost all the cases the variables pushed onto the stack could be accessed using only a signed 8-bit offset present in XA. The function arguments and variables could be moved in and out of the stack in a single PUSH/POP multiple instructions permitted in XA. In fact up to 8 words or 16 bytes of such information could be moved in and out of the XA stack with one instruction, which increases code density to a large extent during procedure calls and context switching. For example, if register variables are in R1,R2,R3, and R4, a single "PUSH R1,R2,R3,R4" instruction will be generated by the XA-compiler. A corresponding function exit will have a "POP R1,R2,R3,R4" for restoring the variables.

All automatic class of variables will be allocated on run-time stack. The XA has full complement of addressing modes on SP to handle dynamic variables in the stack. Table 2 shows some of the XA addressing modes that could be used for such access.

Table 2.

ANSI-C	XA	Comments
SP–>Offset	R+Offset8/16	
*SP	[R]	
SP+	[R+]	Pop

Operators for HLL support

The structure for op-codes of an ideal architecture should be stated in terms of number of operands required and the relationship between the operands. The structure should be oriented toward efficient coding of an instruction that will support programs written in a HLL with minimum compilation. The XA instruction set is designed to handle such efficiency as reflected in Table 3. The set of instructions that supports the general/basic addressing modes are used to describe HLL support in this table.

Table 3. Mapping of XA ALU Operations to C Operators

ANSI C Operator (op)	XA Op-codes
+= , += + C()	ADD, ADDC
–= , –= – C()	SUB, SUBB
< , <= , == , >= , > , != (s/u)	CMP
&= , |= , ^=	AND , OR , XOR

Data movement in C is given by "=" which is the "MOV" instruction in XA. The MOV instruction not only has the general/basic addressing modes, it also has some additional addressing modes for C-code optimization for memory transfer operations like direct-direct, direct-indirect, indirect-autoincrement – indirect-autoincrement.

Table 4 of two operand case **A = A op B** or **B = A op B** is shown below.

Table 4.

ANSI C	XA
C-operations	**Equivalent XA-operations**
R op = R	R, R
R op= *R	R, [R]
R op= *R++	R, [R+]
R op= direct	R, direct
R op= R–>offset	R, [R+offset]
*R op= R	[R], R
*R++ op= R	[R++], R
direct op= R	direct, R
R–>offset op= R	[R+offset], R
R op= constant	R, #constant
*R op= constant	[R], #constant
*R++ op= constant	[R++], #constant
direct op= constant	direct, #constant
R–>offset op= constant	[R+offset], #constant

The three operand cases A = B op C may regularly be translated as:

```
A = B;
A op= C;
```

exception to above is

```
*R++ = B op C is equivalent to
*R = B;
*R++ op = C;
```

Typical/Frequently used C-code **A = B op C** involves operations that will fetch operands from memory, register, and as immediate data which is embedded in the instruction. The XA has the following choices for operand placements for such three operand operations.

Case 1:
If A = register,

then B and C in A = B and A op= C could have the following choices

(i) Register i.e., R = R and R op= R

(ii) Memory i.e., R = Memory and R op= Memory

where Memory = [R] , direct, [R+], [R+Offset]

(iii) Immediate i.e., R = Immediate and R op= Immediate

Case 2:
If A = Memory

where Memory = [R], direct, [R+], [R+Offset]

then B and C in A = B and A op= C could have the following choices:

(i) Register i.e., Memory = Register and Memory op= Register

(ii) Immediate i.e., Memory = Immediate and Memory op= Immediate.

(iii) Memory i.e., Memory = Memory ***([R+], and direct modes only)*** *for B*

The above indicates that virtually all C operations involving two and three operands could be very efficiently translated in XA assembly code (in two operand cases, it is one-to-one) using a cross-compiler.

NULL DETECT/STRING TERMINATOR

Checking for “0” at the end of a string is natural in XA with the MOV instruction. The Z flag is set whenever such a condition occurs. This is especially important in string copy operations where the loop ends whenever a end of string or ‘\0’ occurs which is reflected in the status flag “Z” in XA. The following lists such C-code and equivalent XA instructions.

```
while ((c=getch()) != ‘\0’)      Label:   MOV [R+], memory
buffer[i++] = c;                          BNE      Label
```

Coding Relational Operations

Performing relational evaluation between two operands A and B in C-language involves fetching operands (a) in memory (b) in register or (c) an immediate value, evaluating the condition and then taking appropriate actions which typically involves a branch-if-true or branch-if-false operations

The operand(s) in memory again could be addressed as direct, indirect, indirect-autoincrement, indirect-offset, etc. The XA provides one-to-one translations of such operations.

Typically such C-statements are as follows:

```
if (A cmp_op B)             CMP A, B
{ body} /* true */          Bxx LABEL; branch if false
                            body
                            LABEL:

if (A cmp_op B)             CMP A, B
{ body 1 }                  Bxx L1 ; branch if false
else                        body 1
{ body 2 }                  JMP L2
                            L1: body 2
                            L2:

while (A cmp_op B)          L1: CMP A, B
{ body }                    Bxx L2 ; branch if false
                            body
                            JMP L1
                            L2:
```

Coding Bitwise Operations

C provides 6 operators for bit manipulation. These are & (Logical AND), I (Logical OR), ^ (Logical-XOR), << (Logical Shift-Left), >> (Logical Shift-right), and ~ (one's complement). There is one-to-one equivalence in XA for such operation class:

(a) & – AND,
(b) I – OR, ^ – XOR,
(c) << – ASL,
(d) >> – LSR, and
(e) ~ – CPL.

Compiler Optimization

Some special cases of Multiply and Divide where the multiplier and divisor could be assumed to a power of 2, following translation could be expected from the compiler during optimization which speeds up code execution and make code denser.

Language extensions to XA could be written as the pre-processor macros of the XA C-compiler as shown in Table 5.

Table 5.

C-code	XA code
R *= R	R <<= R
R *= Constant	R <<= Constant
R /= R	R >>= R
R /= Constant	R >>= Constant

ROLC(R,R) – for rotate left through carry, ROL (R,R) and ROL (R, constant) – for rotate lefts, etc. Same holds for ADDC and SUBB also.

Reentrancy

In a multi-tasking or nested interrupt environment, some system or library subroutines may be activated dynamically. These subroutines require duplication of the variable area of the subroutine per each active copy, utilizing essentially *dynamic memory allocation.*

The allocation of the dynamic area is done by a system service call. The dynamic area is allocated either out of the reserved system memory, when large memory exists in the system, or on the stack, when memory is very limited. In the latter case, the stack pointer is adjusted, to reflect the extra bytes reserved. It will be readjusted just prior to returning from the subroutine.

The subroutine code accesses variables using [R+offset] addressing mode. The register is referred to as a Static Base Register or Frame Pointer.

Since the application stack is separate from the interrupt stack, there's no problem with interrupting the dynamic allocation/de-allocation and application stack pointer adjustments.

Floating Point Support

Although the XA does not have a floating point unit, it has special instructions to provide an extensive support for floating point operations. Instructions like NORM (normalize), SEXT (sign extend), ASL, ASR (Arithmetic shifts) and status flag like "N" (sign), all aid in floating point support. Floating point library routines implementing (IEEE or ANSI) floating point provided with compilers could extensively use such instructions for increased code density and throughput in XA.

Dynamic Code Link/Relocability

The XA allows for dynamic code linking through extensive use of FCALL (Far Call 24-bit addressing). This makes code developed for XA highly portable/relocatable in memory.

Simple relocatable code however could use CALL rel16 and CALL [R] addressing modes which is limited to 64K address.

System Interface

When used for RTOS, system mode with its protected features could be extensively used for system management routines/operating System service e.g., *printf etc* and application task switching. This could be easily done in XA through a TRAP # instruction set up by the compiler requesting system service by the application task. In the event of task switching, a system service call sets up the environment for the new task via the resource access privileges of the task, application stack etc.

Author's Acknowledgement

The author recognizes the following Philips Semiconductor XA team members for their review and inputs on this article:

Ata Khan, Ori Mizrahi-Shalom, and Frank Lee

References:

XA User Guide – Philips Semiconductors

XA benchmark versus the architectures 68000, 80C196, and 80C51

AN703

Author: Santanu Roy

BACKGROUND

A computer benchmark is a "program" that is used to determine relative computer core performance by evaluating benchmark execution time by that core. In the brainstorm on microcontrollers for automotive applications, an assembler functional *benchmark for engine management*, which is a typical example of embedded high-end microcontrol, was created. This report gives worked out routines of the functions if they were implemented in assembler language of the compared controllers: Motorola 68000, Intel 80C196, Philips 80C552 and Philips XA. The total execution times of a program "engine cycle" (engine stroke) are calculated and the required program code is estimated for each controller.

Evaluation of performance in a High Level Language (HLL) like C would be preferable, but it is difficult to realize as "the best" compilers for all cores involved then should be used.

This document is generated based on the report number DPE88187. It outlines code density and execution times of the XA, based on most recent information. The execution times are given in terms of clock cycles. Although XA can run at speed of 30 MHz @ 5.0 Volts, for sake of fairness, all cores are evaluated for running at 16.00 MHz. This is reasonable for comparing the cores in the same level of technology.

A separate section is included in this benchmark for "Bit manipulation" function benchmark results only. This (bit-test) routine is a stand alone one and should not be considered as a part of *engine management* routine.

BENCHMARK RESULTS AND CONCLUSIONS

Relative performance on a line

The table below presents the most important result of the assembler benchmark evaluation. It pictures the relative performance of the compared core instruction set on a scale where XA=1.0. Also appended is the performance charts-execution and code density of all the processors.

Total exec.times/core(µs) for all routines (with *occurrences)
5,942 1,560 1089.24 420.84

PERFORMANCE RATIO	8051	68000	80C196	XA
8051	1.0	3.81	5.45	14
68000	0.34	1.0	1.43	3.7
80C196	0.18	0.7	1.0	2.58
XA	0.07	0.27	0.39	1.0

Table 1. XA instruction set execution times and bytes/function

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.94	15.8	18
ADD/SUB	50	0.38	19	4
CMP 24b	13	1.56	20.3	16
CAN 16b	40	0.78	31.2	8
INTPLIN	20	1.98	41.3	14
INTERR	10	6.1	61	41
BRANCH	10		153.1	

XA totals : 350.7 µs
including 20% statistics : 420.84 µs

Table 2. 68000 instruction set execution times and bytes/function

FUNCTION	OC*	68000		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	4.4	52.8	2
FDIV	4	13.4	53.6	16
ADD/SUB	50	2.75	137.5	12
CMP 24b	13	3.2	41.6	14
CAN 16b	40	2.7	108	14
INTPLIN	20	7.5	150	14
INTERR	10	21.9	219	92
BRANCH	10		537.5	

68000 totals : 1,300 µs
including 20% statistics : 1,560 µs

Table 3. 80C196 instruction set execution times and bytes/function

FUNCTION	OC*	80C196		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	1.75	21	3
FDIV	4	9.5	38	19
ADD/SUB	50	1.25	62.5	7
CMP 24b	13	4.25	55.2	14
CAN 16b	40	2.5	100	6
INTPLIN	20	6.4	128	18
INTERR	10	12.8	128	58
BRANCH	10		375	

80C196 totals : 907.7 µs
including 20% statistics : 1,089.24 µs

Table 4. 8051 instruction set execution times and bytes/function

FUNCTION	OC*	8051		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	37.5	450	58
FDIV	4	451.5	1806	96
ADD/SUB	50	7.5	375	19
CMP 24b	13	9.98	129.74	22
CAN 16b	40	9	360	14
INTPLIN	20	25.8	516	20
INTERR	10	31.5	315	70
BRANCH	10		1000	

8051 totals : 4,951.74 µs
including 20% statistics : 5,942 µs

Table 5. Total benchmark execution time results

MICROCONTROLLER CORE	EXECUTION TIME (μs)
XA	420.84
68000	1560
80C196	1089.24
8051	5942

As the total activity has to be completed in one machine stroke of 2 ms, the XA, and the 80C196 will be able to meet the application requirements. The 80C552 originally was assumed to complete the functions over more than one stroke.

Best efficiency is of the XA and the 80C196. The 80C196 includes 3-parameter instructions that reduce the instruction count per function and it has JB/JBN instructions. It also uses half-word (1-byte) codes for frequently used instructions.

The lower code efficiency of the 8051 instruction set can mainly be explained by the "accumulator bottleneck" which is not present in XA: most data has to be transported to and from the accumulator be fore add/sub/cmp can be done, operations on words require 4 "MOV" instructions and 2 data execution instructions. The efficient JB and JBN instructions compensate this for a great part.

BENCHMARK LIMITATIONS

Like all benchmarks, the automotive engine management assembler functional benchmark has some weakness that limit validity of its results.

1. Control in a special (automotive, engine) environment is evaluated.
2. Occurrences of operation overheads are based on estimations.
3. Occurrences of functions are based on estimations.
4. Functions are implemented in assembler, not in a HLL like C.
5. Routines may contain assembler implementation errors.
6. All cores are evaluated at 16.0 MHz

Control in a special environment is evaluated (automotive, engine)

The core performance evaluation is based on a single specialized case. All benchmark implementations are fractions of the automotive engine management PCB83C552 demonstration program.

It can be advocated that the automotive engine control task gives a good example of a typical high demanding control environment, where many >= 16 bit calculations have to be done.

Occurrences of overheads are based on estimations

The assembler functional benchmark is not a full implementation of a program. Arbitrary choosing location for storage of parameters in register file or (external) memory, for instance, has for some instruction set a considerable effect on the total execution time.

For the different core parameter storage is chosen where possible using the core facilities to have minimum access overhead.

Occurrences of functions based on estimations

Occurrences is estimated on basis of experience of the automotive group. In a real implementation of an engine controller accents may shift. As most functions already include some "instruction mix", the effect of changes in occurrences is limited.

Functions are implemented in assembler, not in a HLL like C

Control programs for embedded systems get larger, have to provide more facilities and have to be realized in shorter development times. The only way to do this is to program in a HLL like C. Efficient C-language program implementation requires different features from microcontrollers than assembly programs. Results of this assembler benchmark evaluation therefore have a restricted value for ranking microcontroller performances for future HLL applications.

Benchmark ranking on basis of HLL like C requires good C-compilers of all the devices involved are needed. The quality of the C-compilers really has to be the best there is: HLL benchmarking measures not only the micro characteristics, but even more the compiler ability to use these qualities. As these are not available for all the micros evaluated, all routines are worked out only in assembly.

Routines may contain assembler implementation errors

Assembler routine implementations are made after a short study of the micro specifications and are not checked by assembling or debugging in real hardware environment.

It can be rather safely said that a complete system setup and program debug to correct errors would not lead to considerable differences in performance results. Deviations in function occurrences and overheads may have a more significant effect on performance ratios.

All cores are evaluated at 16.0 MHz

A 16.0 MHz internal clock frequency seems a reasonable choice for comparing the cores at the same level of technology.

ASSEMBLER FUNCTIONAL BENCHMARK FOR AUTOMOTIVE ENGINE MANAGEMENT

This benchmark is a functional benchmark: it is a collection of functions to be executed in an automotive engine management program. It would be preferable to implement the complete control program in assembler and evaluate it in a real hardware environment, but this is not practical as every implementation requires many man-months to realize.

To implement the assembly functional benchmark for automotive engine management correctly the "rules and details" described in this section have to be followed carefully.

The assembler functional benchmark embraces all activity to be completed in 1 program cycle that corresponds with 1 engine stroke of 2 ms. The benchmark execution time will be calculated as the sum of the products of functions and their occurrence rates in 1 calculation cycle.

Branches are evaluated separately as "branch penalties" have considerable effect of program execution efficiency. Estimated (branch count)*(average branch time) is added to the function execution times.

The relative estimated overhead for statistics does not contribute to the evaluation of speed performance ratios, but they have to be considered when looking at the total execution time required / engine stroke cycle. therefore the real total execution time is multiplied with the statistics overhead factor (1.2*).

NO.	FUNCTION DESCRIPTION	OCCURRENCES
1	16×16 Multiply	12
2	Floating Point divide (16:16)	4
3	Add/Subtract (24)	50
4	Compare (24)	13
5	CAN cmp/mov 10*8	80
6	Linear Interpolation (8*8)	20
7	Interrupts	10
8	Program control branches	500
9	Statistics (20%)	1.2 *

FUNCTION PARAMETER ALLOCATION

Most functions are very short in exec. time, so that the function parameter data access method has great effect on the total time. Thus it is to be considered carefully.

Some core features a large register files (XA, 80C196) in which variables can be stored, others with few registers (68000) have to store all data in memory.

For the XA/80C196 processor, data stored in the lower part of register file, or in SFRs for I/O, can be accessed using "direct" addressing, but table data, used, e.g., for 3 bye compare, is stored in "external memory".

The 68000 assume data in memory (or memory mapped I/O) as not enough data registers are available. All 68000 memory data has to be accessed using long-absolute addressing: 68000 short addresses are relative to memory address 0000 and are therefore not useful.

For more complex functions 16*16 multiply, Floating point division and interpolation, data is assumed to be already in registers.

16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32-bit result written into a register pair.

Divide (16:16) "floating point"

The floating point division is entered with parameters in registers:

a divisor, a dividend and an "exponent" that determines the position of the fraction point in the result.

Floating point binary 16/16 division is a function that is normally not included in HLL compilers as it requires separate algorithms for exponent control and accuracy is limited. For assembler control algorithms, floating point division can be quite efficient as it is much faster than normal "real" number calculations (where no "floating point accelerator" hardware is available).

Compare 24-bit variables

Note that 24-bit compare is very efficient for "real" 16-bit and 8-bit) controllers, but for automotive engine timers, 24-bit seems a good solution.

Compare must give possibility to decide >, < or =. For 68000, and 80C196 instruction set LT, EQ and GT are included in the cc after CMP.

CAN move and compares

For service of the CAN serial interface, it is estimated that 40* (2 byte compares + branch) have to be done. Devices with 16-bit bus assumes word access. An average branch is included in the CAN compare function.

Linear Interpolation (8*8)

The interpolation routine is entered with 3 register parameters:

1. Table position address
2. X fraction
3. Y fraction

The routine first interpolates using the X fraction the values of F(x.x, y) between F(x,y) ....V(x+1, y) and of F(x.x, y+1) between F(x, y+1) .... F(x+1, y+1). From F(x.x, y) and F(x.x, y+1) the value of F(x.x, y.y) is interpolated using the fraction of y.

The table is organized as 16 linear arrays of 16 x-values, so that an V(x,y) can be accessed with table origin address +x+16*y = "Table Position Address". In x-direction the interpolation can be done between the "Table Position" value and next position (+1). Interpolation in y-direction is done by looking at "Table Position" + 16.

For linear interpolation time the 2-dimensional interpolation time and byte count are divided by 3 to include some "overhead" into linear interpolation.

Interrupts

The average interrupt routine overhead includes the following stages:

a. Interrupt recognition and return

b. 1 * (long) branch

c. 2 * jump (short) on bit

d. 1 * call (long) and subroutine return

e. 2* set bit and 2 * clear bit

f. 5 * POP and 5 * PUSH (or move multiple) [free 5 registers for local use]

g. 1 * mov #xxx, PST

Program Control Overheads

For a given algorithm, the Program Control Overheads consisting of a number of decisions (branches) and subroutine calls is independent of the instruction set used, except for cases where functions can be replaced by complex instructions. The most important exception cases, MPY words and Floating Point Division are handled in this benchmark separately.

Most 16-bit cores use more pipeline stages so that taken branches add branch time penalty for these CPU's due to pipeline flush. This effect can be found in the branch execution time tables.

More efficient data operations and pipeline penalty of the more complex instruction set of 16-bit cores lead to considerable higher relative time used for branch instructions.

To incorporate the influence of branches in the benchmark the number of branches to be included must be estimated. For byte and bit routines, branches occur more frequent. Average branch time of 25% may be a good guess. For the automotive engine management benchmark that executes in approx. 5000/µS (on 8051) results in +/- 1250 /µS or 625 branches. As a part of the branches already taken account for in the compare functions the number of additional program control branches is estimated 500 branches.

To estimate the average branch execution time, an estimated relative occurrence of the branch types has to be made.

Table 6. Estimated relative occurrence of the branch types

	TYPE	RELATIVE	ABSOLUTE OCCURRENCE
Absolute Jumps	AJMP/JMP	20%	100
Subroutine calls	ACALL/JSR	20%	100
Jump on condition (rel)	Bcc/Jcc	40%	200
Jump on bit (rel)	JB/JBN	20%	100

Statistic Routine Overheads

Statistic routines are estimated as relative program overheads, only to get an indication of the required total processing time in a real engine management application. "Statistics" are mainly arithmetic routines to determine table corrections. They use about 20% of the total time.

XA BENCHMARK RESULTS

The following analysis assumes worst case operation. At any point in time, only 2 bytes are available in the instruction Queue. An instruction longer than 2 bytes requires additional code read cycle.

APPENDIX 1

XA Function Implementations

XA reference: *XA User's Manual 1994*

16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32-bit result written into a register pair. The MUL.w R,R is encoded in the XA instruction set as a 2 byte instruction. The exact optimization for this instruction (such as skip over 1's and 0's) has not been concluded at this point, and the execution time may be data dependent and shorter than one outlined here.

The basic algorithm utilizes 2-bit Booth recoding. Instruction fetch and Decode time overlaps the execution of the preceding instruction (except when following a taken branch), so it is ignored. The total execution time is either 11 or 12 clocks, including operand fetch and write back (1 clock is dependent on critical path analysis).

A1.1: 16×16 Multiply

```
                                                         Bytes   Clocks
        MUL.w   R0, R1                                     2       12 (0.75 µS)
```

A1.2: Floating Point 16x16 Divide:

The algorithm here follows the one outlined for the 80C196.

```
Arguments:      R4 = Dividend (extend into R5 for 32 bits)
                R6 = Divisor Mantissa
                R0 = Divisor  Exponent

                                                         Bytes   Clocks
FPDIV:
        ADDS    R6, # 0              ; Add short format      2      3
        BEQ     L1                   ; Check for DIVBY0      2      3  (not taken)
                                     ;
SGNXTD_AND_SHFT:                     ;
        SEXT    R5                   ; Sign extend into R5   2      3
        ASL     R4, R0               ; 13 position shifts    2      11
                                     ;
DIV:                                 ;
        DIV.d   R4, R6               ; Divide 32x16 signed   2      21
        BOV     L1                   ; Branch on Overflow    2      6  (taken)
        RET                          ; Normal termination    2      8
                                     ;
L1:                                  ;
        MOVS    R4, # -1             ; Overflow - Max Result 2      3  (not  executed)
        RET                          ;                       2      8

                                                            18      63 (3.94 µS)
```

A1.3: Extended 32-bit subtract

```
        SUB     R4, R2                                     2       3
        SUBB    R5, R3                                     2       3

                                                           4       6  (0.38 µS)
```

A1.4: Compare 24-bit Variables

Only minimum execution time is considered here. An average branch is included after compare.

```
                                                        Bytes   Clocks
MOV.w       R1, sfr1     ;                                3       4
MOV.w       R0, sfr0     ; read the two SFR's into        3       4
                         ; registers
CMP.b       R1, mem1     ; direct addressing              3       4
BNE         L1           ; average (6t/3nt)               2       4.5
CMP.w       R0, mem2     ;                                3       4
Bxx         LABEL?       ; average                        2       4.5

                                                          16      25 (1.56 µS)

xx -> GT or LT or EQ
```

A1.5: CAN Move and Compare

Application:

For service of CAN (Controller Area Network) serial Interface it is estimated that 40* (2 byte compares + branch) have to be done. One parameter is in SFR, the other in internal memory. Again, minimum execution times are considered.

```
                                                        Bytes   Clocks

MOV         R0, sfr0     ;                                3       4
CMP         R0, mem0     ;                                3       4
Bxx         LABEL?       ; average                        2       4.5

                                                          8       12.5 (0.78 µS)
```

A1.6: Linear Interpolation

```
Arguments:
          R0 = Table Base (assumed < 400 Hex)
          R2 = Fraction 1
          R4 = Fraction 2
          R6 = Result

                                                        Bytes   Clocks
LIN_INT:
          MOV        R6, [R0+]          ;                 2       4
          MOV        R1, [R0]           ;                 2       3
          SUB        R1, R6             ;                 2       3
          MULU.w     R6, R2             ;                 2       12
          MOV.b      R1H, R1L           ;                 2       3
          MOVS.b     R1L,#0             ;                 2       3
          ADD        R6, R1             ;                 2       3
          ADD        R0, #15            ;                 2       3
          MOV        R1, [R0+]          ;                 2       4
          MOV        R5, [R0]           ;                 2       3
          SUB        R5, R1             ;                 2       3
          MULU.w     R5, R2             ;                 2       12
          MOV.b      R1H, R1L           ;                 2       3
          MOVS.b     R1L,#0             ;                 2       3
          ADD        R1, R5             ;                 2       3
          SUB        R1, R6             ;                 2       3
          MULU.w     R1, R4             ;                 2       12
          MOV.b      R1H, R1L           ;                 2       3
          MOVS.b     R1L,#0             ;                 2       3
          ADD        R6, R1             ;                 2       3
          RET                           ;                 2       6

                                                          42      95 (5.94 µS)
Linear Interpolation (2 dim. time / 3) = 14 bytes, 1.98 µS
```

A1.7: Interrupt Overhead

Note: Interrupt overhead, as defined in the benchmark, applies to performance calculations. It does not consider the interrupt latency associated with completing the current instruction.

All transfers are to / from internal memory, all addresses are 16-bit long.
{
Saves 2 words on stack = 4 clks
Prefetching ISR = 3 clks
Overhead through Interrupt Controller = 3 clks (allow synch + avoid metastability)
i.e., total = 10 clks
}

```
Interrupt Accept/Return                                      0/2      10+8
JMP      rel16               ; uncond. x 2                   3x2      6x2
Bxx      bit, rel8           ; Branch on bit test x 2        2x2      4.5x2
CALL     rel16               ; Long Call (PZ assumed)        3        4
RET                          ; Subroutine return             2        6
SETB     bit                 ; Set bit x 2                   3x2      4x2
CLR      bit                 ; Clear bit  x 2                3x2      4x2
PUSH     Rlist (5)           ; 5 PUSH Multiple               2        15
POP      Rlist (5)           ; 5 POP Multiple                2        12
MOV      PSWL, #data8        ; imm. byte to PSWL             4        3
MOV      PSWH, #data8        ; needs 2 for 8-bit sfr         4        3
                             ; bus
                                                             41       98 (6.1 µS)
```

A1.8: Program Overhead

Branches are assumed taken 70% of the time, all addresses are external. Code is assumed a run-time trace, code size cannot be calculated; based on the same approach taken for 80C196, code size is 1400 bytes.

```
JMP      rel16          ; Long branch  x 100               3x100    6 x 100
CALL     rel16          ; Call  x 100 (Page 0)             3x50     4 x 50
RET                     ; Subroutine return x 100          2x100    6 x 50
Bxx      rel8           ; Condl. short branch x 100        2x200    4.5 x 200
JB/JNBbit, rel8         ; Bit test & branch x 100          2x100    4.5 x 100

                                                           1400      2,450
                                                                    (153.1 µS)
```

A1.9: XA TOTALS

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.94	15.8	18
ADD/SUB	50	0.38	19	4
CMP 24b	13	1.56	20.3	16
CAN 16b	40	0.78	31.2	8
INTPLIN	20	5.94	118.8	42
INTERR	10	6.1	61	41
BRANCH	10		153.1	

Conclusion:

An assumption is made that XA code is in first 64K (PZ) as the 80196 has a 64K address space only.

APPENDIX 2

8051 Function Implementations

8051 reference: *Single chip 8-bit microcontrollers PCB83C552*
Users manual 1988

A2.1: 80C51 Multiply 16×16

The 80C51 core performs 8-bit multiply only. A 16×16 multiply has to be done by splitting X and Y into XH, XL and YH, YL so that:

P3..P0 = (XH*256+XL)*(YH*256+YL) =
XH*YH*65536+(XH*YL+XL*YH)*256+XL*YL

```
                                 Clocks  Bytes
MPY:
        MOV     R1,XH              2       3
        MOV     R2,XL              2       3
        MOV     R3,YH              2       3
        MOV     R3,YL              2       3
        MOV     A,R2               1       1        ;XL
        MOV     B,R4               1       3        ;YL
        MUL     AB                 4       1
        MOV     P0,A               1       2        ; Lowest multiply result byte
        MOV     A,R4               1       1        ;YL
        MOV     R4,B               2       3        ; XL*YL upper byte (*256)
        MOV     B,R1               2       3        ;XH
        MUL     A,B                4       1        ;XL*YL
        ADD     A,R4               1       1
        MOV     R4,A               1       1        ;upper (Xl*YL)+lower(XH*YL) in R2
        MOV     A,B                1       2
        ADDC    A,#0               1       2
        XCH     A,R2               1       1        ;XL upper (XH*YL) in R2
        MOV     B,R3               3       2        ;YH
        MUL     A,B                4       1        ;XL*YH
        ADD     A,R4               1       1
        MOV     P1,A               1       2
        MOV     A,B                1       2
        ADDC    A,R2               1       1
        MOV     R2,A               1       1
        MOV     A,R3               1       1
        MOV     B,R1               2       3
        MUL     AB                 4       1
        ADD     A,R2               1       1
        MOV     P2,A               1       2
        MOV     A,B                1       2
        ADDC    A,#0               1       2
        MOV     P3,A               1       2

        Total                     50      58
```

50 clocks = 50*12 = 600 clocks (37.5 µs @ 16.0 Mhz)

8051 MPY 16×16 (MPY Bytes) 50 clocks = 37.5 µs / 58 bytes

A2.2: 8051 Divide (16/16) "floating point"

Divide (R6, R7) (dividend) by (R4,R5) (divisor) with (R0) bits after the fraction point.

Alignment of MSBits of operand in R6.7 and R4.7 using R0 as bit counter.

```
                             Clocks  Bytes
FDV:
        INC     R0              1       1
        INC     R0              1       1
        MOV     R3,#0           1       2
        MOV     R2,#0           1       2
        CLR     C               1       1
        CLR     F0              1       2
        MOV     A,R4            1       1
        JB      ACC.7, L2       2       3
        JNZ     L1              2       2
        MOV     A,R5            1       1
        JZ      LX              2       2
L1:
        MOV     A,R5            1       1
        RCL     A               1       1
        MOV     R5,A            1       1
        MOV     A,R4            1       1
        RCL     A               1       1
        MOV     R4,A            1       1
        INC     R0              1       1
        JNB     ACC.7, L1       2       3
L2:
        MOV     A,R6            1       1
        JB      ACC.7, L6       2       3
L3:
        MOV     A,R7
        RLC     A               1       1
        MOV     R7,A            1       1
        MOV     A,R6            1       1
        RLC     A               1       1
        MOV     R6,A            1       1
        DJNZ    R0, $+4         2       2
        AJMP    LX              2=0     3
        JNB     ACC.7,L3        2       3
        AJMP    L6              2       3
L4:
        MOV     A,R3
        RLC     A               1       1
        MOV     R3,A            1       1
        MOV     A,R2            1       1
        RLC     A               1       1
        MOV     R2,A            1       1
        JNC     L5              2       2
        MOV     R2,#0FFH        1       1
        MOV     R3,#0FFH        1       1
        SJMP    LX              1       1
L5:
        CLR     C               1       1
        MOV     A, R7           1       1

        RLC     A               1       1
        MOV     R7,A            1       1
        MOV     A,R6            1       1
        RLC     A               1       1
        MOV     R6,A            1       1
        JNC     L5              1       1
        MOV     F0,C            1       2
```

```
L6:
        CLR     C               1       1
        MOV     A,R7            1       1
        SUBB    A,R4            1       1
        JNC     L7              2       2
        JNB     F0,L8           2       3
        CPL     C               1       1
L7:
        MOV     R6,A            1       1
        MOV     A, R1           1       1
        MOV     R7,A            1       1
L8:
        CPL     C               1       1
        DJNZ    R0,L4           2       2
        MOV     A,R3            1       1
        ADD     A,#0            1       1
        MOV     R3,A            1       1
        MOV     A,R2            1       1
        ADD     A,#0            1       2
        MOV     R2,A            1       1
LX:
        RET                     2       1
```

Total 96 bytes 13 branch instructions (=35 bytes== 36%)

```
Timing : 3 divide cases :               subtracts       shifts          total           average
1. R0=0E, 8-bit/14 bit          -->     15-8+2=9        8+2=9           32 subtracts    11
2. R0=08, 12-bit/14 bit         -->     8-4+4=8         4+4=8           17+11 shifts    6+4
3. R0=10, 11-bit/12 bit         -->     16-5+4=15       5+5
17+4*9+6*10+(15.5+10*31.5)+8=451.5 clocks = 338.6 µS
```

8051 UFDIV 16/16 (sub/sft) : 338.6 clocks = 451.5 µs, 96 bytes.

A2.3: 8051 Add/Sub

```
                                Bytes   Clocks
ADS:
        CLR     C               1       1
        MOV     A,X0            1       2
        SUBB    A,Y0            1       2
        MOV     Z0,A            1       2
        MOV     A,X1            1       2
        SUBB    A,Y1            1       2
        MOV     Z0,A            1       2
        MOV     A,X2            1       2
        SUBB    A,#0            1       2
        MOV     Z2,A            1       2
                                10      19
```

8051 ADD/SUB in reg file 10 clocks = 7.5 µs, 19 bytes

8051 CMP enabling JZ JNZ JC JNC

The 8051 decisions made with branches are one of these three :

```
        JC      lt              2       2
        JC                      2       2
        JZ      eq              2       2
        JC                      2       2
        JNZ     gt              2       2
```

8051 compare decision branches take average : 10/3 clocks => 2.5 µs

A2.4: 8051 CMP 3 byte compare

```
                              Bytes  Clocks
CM3:
       CLR     C                1      1
       MOV     A,X2             1      2
       SUBB    A,Y2             1      2
       MOV     R0,A             1      2
       MOV     A,X1             1      2
       SUBB    A,Y1             1      2
       ORL     R0,A             1      2
       MOV     A,X2             1      2
       SUBB    AY2              1      2
       Orl     A,R0             1      2
       Jcc     xxxx             3.33   3.33

                                10     19
```

8051 CMP 3 byte data in reg file 13.3 clocks = 9.975 µs, 22.3 bytes

A2.5: 8051 2-byte CAN compares

```
                              Bytes  Clocks
CAN:
       MOV      DPTR,aX1        2      3           ; one compare src in X-RAM
       MOVX  A, @DPTR           1      2
       CJNE     A,Y1            1      2
       MOV      DPTR,aX2        1      2           ; one compare src in X-RAM
       MOVX  A,@DPTR            1      2
       CJNE     A,Y2            2      3

                                12     14
```

8051 CAN CMP XRAM/Direct 9 µs, 14 bytes

A2.6: 8051 2-dimensional interpolation

```
At the start registers are prepared
A       : position in table (x+16*y)
DPTR    : Start address of table (aligned at 256 byte boundary)
R0      : x-fraction   R1      : y-fraction
Result  : ACC registers used : ACC,R0,R1,R2,R4,R5,R6

                                 Clocks  Bytes
INT:
        MOV     DPL,A              1       2               ;POS X,Y
        ACALL GVAL                 2       2
        MOV     R4,A               1       1
        MOV     A,DPL              1       2
        ADD     A,#15              1       2
        MOV     DPL,A              1       2
        ACALL   GVAL               2       2
        MOV     REG6,R4            1       2
        MOV     B,R1               1       2
        ACALL INTP                 1       2
        RET                        2       1
GVAL:
        MOVX A,@DPTR               2       1
        MOV     R6,A               1       1
        INC     DPL                2       1
        MOVX A,@DPTR               2       1
        MOV     B,R0               1       2

INTP:
        CLR     SF                 1       2
        CLR     C                  1       1
        SUBB    A,R6               1       1
        JNC     INT1               2       2
        CPL     A                  1       1
        INC     A                  1       1
        SETB    SF                 1       2
INT1:
        MUL     A,B                4       1
        XCH     A,B                1       2
        CLR     C                  1       1
        RRC     A                  1       1
        XCH     A,B                1       2
        XCH     A,B                1       2
        CLR     C                  1       1
        RRC     A                  1       1
        XCH     A,B                1       2
        JB      SF,INT2            2       3
        ADDC    A,R6               1       2
        RET                        2       1
INT2:
        XCH     A,R6               1       2
        SUBB    A,R6               1       2
        RET                        2       1
```

Total 2-dim. interpolation : 15+2*(8+24)+24=103 clocks = 77.25 µs, 59 bytes

8051 Linear interpolation : (2–dim. intp time /3) = 103/3 =25.75 µs, 20 bytes

A2.7: 8051 Interrupt Overhead

		Bytes	**Clocks**
a.	interrupt	2	2 (vector)
	RETI	2	1
b.	AJMP 2*	4	4
c.	JB 2*	4	6
d.	ACALL	2	2
	RET	2	1
e.	SETB 2*	2	4
	CLRB 2*	2	4
f.	POP 5*	10	10
	PUSH 5*	10	10
g.	MOV 1*	2	2
		42	**46**

8051 Interrupt Overhead 42 clocks = 31.5 µs

A2.8: 8051 Program Overhead

TYPE	OCCURRENCE	8051		BYTES	
LJMP/JMP	100	2	200	3	300
LCALL/JSR	100	2	200	3	300
Jcc/Bcc	200	2	400	3	600
JB/JBN	100	2	200	3	300
total cylces µsec			1000 750		1500

A2.9: 8051 Totals

FUNCTION	OC*	8051	
		EXEC	*OC
1. MPY	12	37.5	450
2. FDIV	4	338.6	1354.4
3. ADD/SUB	50	7.5	375
4. CMP 24b	13	9.98	129.74
5. CAN 16b	40	9	360
6. INTPLIN	20	25.8	516
7. INTERR	10	31.5	315
8. BRANCH	10		750

8051 totals : 4250.14 µs
including 20% statistics : 5,100.2 µs

APPENDIX 3

68000 implementations

68000 reference: *SC68000 microprocessor users manual*

(Motorola copyright; Philips edition 12NC: 4822 873 30116)

A3.1: 68000 16x16 Multiply

The 68000 can use 1 <ea> with MUL and move a long word result.

```
MUL      R0,R1                    2        70
```

total: 4.375 µs, 2bytes

A3.2: Floating point division 16:16

```
(R0) Accuracy,          (R1)/(R2)    R1 result

                                 Bytes   Clocks
FDV:
          EXT.l    R1              2        4
          TST      R2              2        4
          BEQ      L1              2        10/8
          ASL      R0,R1           2        32
          DIVU     R2,R1           2        140
          BVC      L2              2        10/8
L1:
          MOVI     #-1,R1          2        4
L2:
          RTS                      2        16
```

total : 214 clocks or 13.375 µs, 16 bytes

A3.3: Add/Sub

```
                                 Bytes   Clocks
ADDS:
          MOV.l    A,R0            6        20
          ADD.l    R0,C            6        48
```

total : 44 clocks or 2.75 µs, 12 bytes

A3.4: Compares 24 (=32) bit

```
                                 Bytes   Clocks
CMP1:
          MOV.l    X,R0            6        20
          CMP.l    Y,Rn            6        22
          BLT/EQ/GT (av) 2         9
```

total : 51 clocks or 3.19 µs, 14 bytes

A3.5: CAN move and compares (16-bit)

```
                                 Bytes   Clocks
CMPw:
          MOV.w    X,R0            6        16
          CMP.w    Y,Rn            6        18
          BLT/EQ/GT (av)           2        9
```

total : 43 clocks or 2.69 µs, 14 bytes

A3.6: 2-dimensional interpolation

A0 : table position, R0 : fraction1, R1 : fraction 2 , R2 : result, R3, R4

```
                                            Bytes   Clocks
CMPw:
           MOV.w          (A0), R2            2        8
           ADDQ.l         #1,A0               2        8
           MOV.l          (A0), R3            2        8
           SUB.w          R2,R3               2        4
           MULu           R0,R3               2        74
           ASR.l          #8,R3               2        28
           ADD.w          R3,R2               2        4
           ADDI.l         #15,A0              4        8
           MOV.w          (A0),R3             2        8
           ADDQ.l         #1,A0               2        8
           MOV.w          (A0),R4             2        8
           SUB.w          R3,R4               2        4
           MULu           R0,R4               2        74
           ASR.l          #8,R4               2        28
           ADD.w          R4,R3               2        4
           SUB.w          R2,R3               2        4
           MULu           R1,R3               2        40
           ASR.l          #8,R3               2        22
           ADD.w          R3,R2               2        4
           RTS                                2        16
```

total : 362 clocks or 22.62 µs, 42 bytes

Linear interpolation is 2-dim. interpolation /3 :

1-dim. interpolation 7.54 µs, 14 bytes

A3.7: 68000 Interrupt Overhead

```
                                 Clocks  Bytes
a.         interrupt               44      4
           RETI                    20      2
b.         JMP 2*                  24      24
c.         BTST+BNE 2*             60      16
d.         BSR                     18      4
           RTS                     16      2
e.         BSET/BCLR 4*            96      24
f.         MOVEM 2* n=5            64      12
g.         MOVI     #xx,CCR        8       4
                                   350     92
```

68000 INTerrupt overhead 350 clocks = 21.87 µs, 92 bytes

A3.8: 68000 Program Overhead

For the 68000, the JB/JBN branches have to be constructed :

```
                                        Clocks  Bytes
MOV.w           ABS.l,Rn                  12      6
ANDI.w          #bitmask,Rn               8       4
BEQ/BNE         rel.address               10      2
```

total JB/JNB execution : 34 clocks, 12 bytes

Now the absolute (estimated) branch time can be calculated, taking the core difference in account.

TYPE	OCCURRENCE	68000		BYTES	
LJMP/JMP	100	12	1200	6	600
LCALL/JSR	100	20	2000	8	800
Jcc/Bcc	200	10	2000	2	400
JB/JBN	100	34	3400	12	1200
total cycles µsec			8600 537.5		3000

A3.9: 68000 Totals

FUNCTION	OC*	68000	
		EXEC	*OC
1. MPY	12	4.4	52.8
2. FDIV	4	13.4	53.6
3. ADD/SUB	50	2.75	137.5
4. CMP 24b	13	3.2	41.6
5. CAN 16b	40	2.7	216
6. INTPLIN	20	7.5	150
7. INTERR	10	21.9	219
8. BRANCH	10		537.5

68000 totals : 1,300 µs
including 20% statistics : 1,560 µs

APPENDIX 4

80C196 function implementations

80C196 reference: *Embedded controller handbook* vol II-16 bit

Copyright : Intel Corp.

A4.1: 80C196 Unsigned multiply P=X*Y (16x16)

```
                                Bytes  Clocks
MUL           R0,R1               3       28
```

total: 1.75 µs, 3 bytes

A4.2: Floating point division 16:16

```
(R0) Accuracy, (R4)/(R8)   R4 result

                                Bytes  Clocks
FDV:
        EXT   R4                  2       4
        AND   R8,#FFFF            4       5
        JE    L1                  2       8/4
        SHLL  R4,R0               3       20
        DIVU  R8,R4               3       24
        JNV   L2                  2       4/8
L1:
        LD    R4,#FFFF            2       5
L2:
        RET                       1       11
```

total: 76 clocks or 9.5 µs, 19 bytes

A4.3: Add/Sub

```
                                Bytes  Clocks
ADDS:
        SUB   R5,R1,R3            3       5
        SUBB  R4,R0,R2            4       5
```

total : 10 clocks or 1.25 µs, 7 bytes

A4.4: 80C196 “3-byte compare”

```
                                Bytes  Clocks
        CMP   Rn,Y1               5       9
        BNE   L1                  2       4/8
        CMP   Rm,Y2               5       9
L1:
        BLT/EQ/GT  (av)           2       4/8
```

Average total: 34 clocks or 4.25 µs, 14 bytes

A4.5: CAN move and compares (16-bit)

```
                                Bytes  Clocks
        CMP   Rx,Y                4       9
        BLT/EQ/GT (av)            2       6
```

total : 15 clocks or 2.5 µs, 6 bytes

A4.6: 80C196 2-dimensional interpolation using in-line linear interpolations

R0 : table position, R2=fraction1, R4=fraction2, R6=result, R8, R10

```
                                   Bytes  Clocks

      LD            R6,[R0]+         3       6
      LD            R8,[R0]+         3       5
      SUB           R8,R6            3       4
      MULU          R8,R2            3       14
      SHRAL         R8,#8            3       15
      ADD           R6,R8            3       4
      ADD           R0,#15           4       6
      LD            R8,[R0]+         3       6
      LD            R6,[R0]          3       5
      SUB           R10,R8           3       4
      MULU          R10,R2           3       14
      SHRAL         R10,#8           3       15
      ADD           R8,R10           3       4
      SUB           R8,R6            3       4
      MULU          R8,R4            3       14
      SHRAL         R8,#8            3       15
      ADD           R6,R8            3       4
      RET                            1       14
```

total : 153 clocks or 19.1 µs, 53 bytes

Linear interpolation is 2-dim. interpolation /3 :

1-dim. interpolation 6.4 µs, 18 bytes

A4.7 80C196 Interrupt Overhead

```
                              Clocks  Bytes
a.      interrupt /RTE          27      2
b.      LJMP 2*                 14      6
c.      JB      2*av.7          14      6
d.      CALL/RTS                22      4
e.      BSET/BCLR 4*            28      16
f.      POP 5*                  40      10
        PUSH    5*              55      10
g.      MOVI    #xx,CCR         5       4

                                205     58
```

80C196 INTerrupt overhead 205 clocks = 12.8 µs, 58 bytes

A4.8: 80C196 Program Overhead

TYPE	OCCURRENCE	68000		BYTES	
LJMP	100	7	700	3	300
LCALL/RET	100	22	2200	4	400
Jcc/Bcc	200	7	1400	2	400
JB/JBN	100	7	700	3	300
total cycles µsec			6000 375		1400

80C196 totals : 958.1 µs
including 20% statistics : 1150 µs

FUNCTION	OC*	80C196	
		EXEC	*OC
1. MPY	12	1.75	21
2. FDIV	4	9.5	38
3. ADD/SUB	50	1.25	62.5
4. CMP 24b	13	4.25	55.2
5. CAN 16b	40	1.88	150.4
6. INTPLIN	20	6.4	128
7. INTERR	10	12.8	128
8. BRANCH	10		375

BIT MANIPULATION

Copy a bit from one location to another in memory. Complement the bit in the new location

Note: Assumed that memory is on-chip and directly addressed.

Bit "x" of mem0 needs to be copied to bit "y" of mem1.

XA

```
        CLR     C               ; clear Carry                      3       4
        ORL     C, /bitm        ; compl. bit and save in C         3       4
        MOV     bitn, C         ; move mem0.x -> mem1.y            3       4

                                                                   9      12
                                                                      (0.75 µS)
```

Intel 80C196

Note : States = clock (period)/ 2

Move complement of bit "m" to "n" in memory

R3 = memory byte having bit "m"
R4 = memory byte having bit "n"
R0 = Used as bit-mask register
R1 = position of "m" in mem0
R2 = position of "n" in mem1

```
                                                              Bytes   States
        LD      R0, 1                   ; Load 1 in Reg
        SHLB    R0, R2                  ; position of bit "n     3      16
                                        ; in R2
        NOTB    R0                      ; complement             2      4
        JBC     R3,bitm, L1             ; test bit "m" polarity  3      7 (av)
        ANDB    R4, R0                  ; reset "n" if "m" = 0   3      4
L1:
        ORB             R4, R0          ; set "m" otherwise      3      either/or

                                                                 14     31 (3.88 µS)
```

Motorola 68000

```
                                                              Bytes   States
        BTST    bitm                    ; Test bit               2      4
        BEQ     L1                      ; Branch if reset        2      6
        BCLR    bitn                    ; Test bit and clear (~m = 0) 2 4
        .......
        .......
L1:     BFSET   bitn                    ; Test bit and set (~m = 1)  2  either/or

                                                                 8      14  (0.88 µS)
```

8051 Bit-test

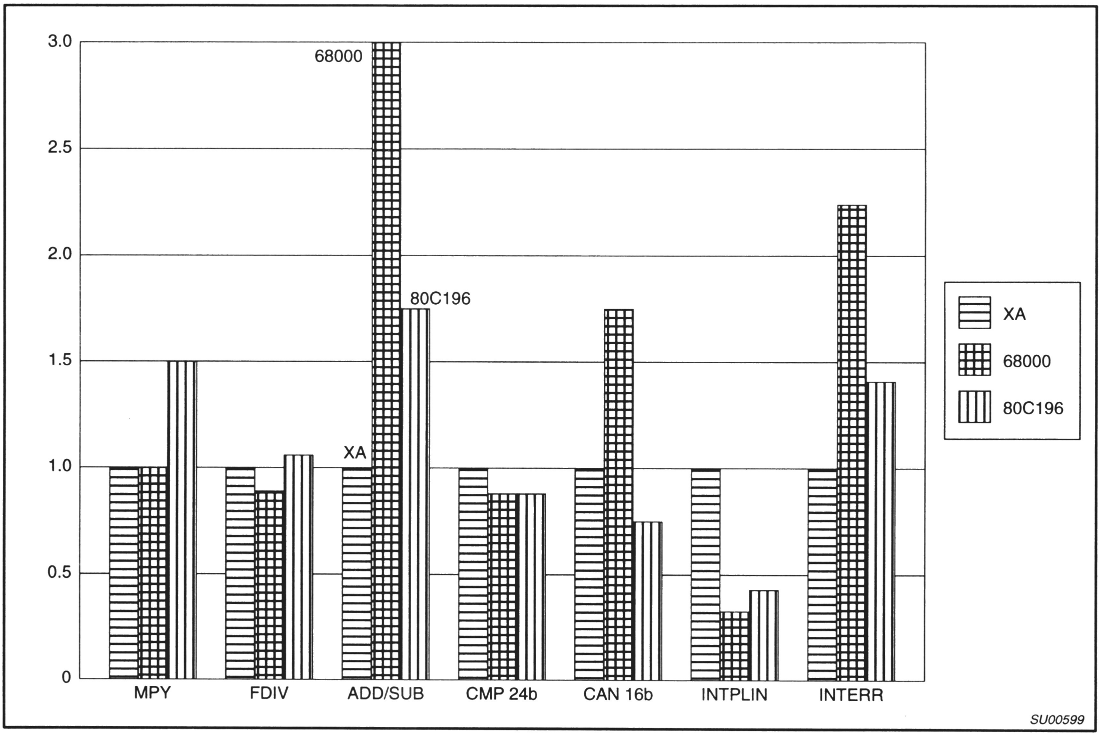
```
        MOV     C, bitm                                          2      12
        CPL     C                                                1      12
        MOV     bitn, C                                          2      24

                                                                 5      48 (3.0 µS)
```

XA CODE DENSITY RESULTS

Graph showing performance with respect to 68000, and 80C196 cores normalized with respect to XA. The 80C51 is included just for reference.

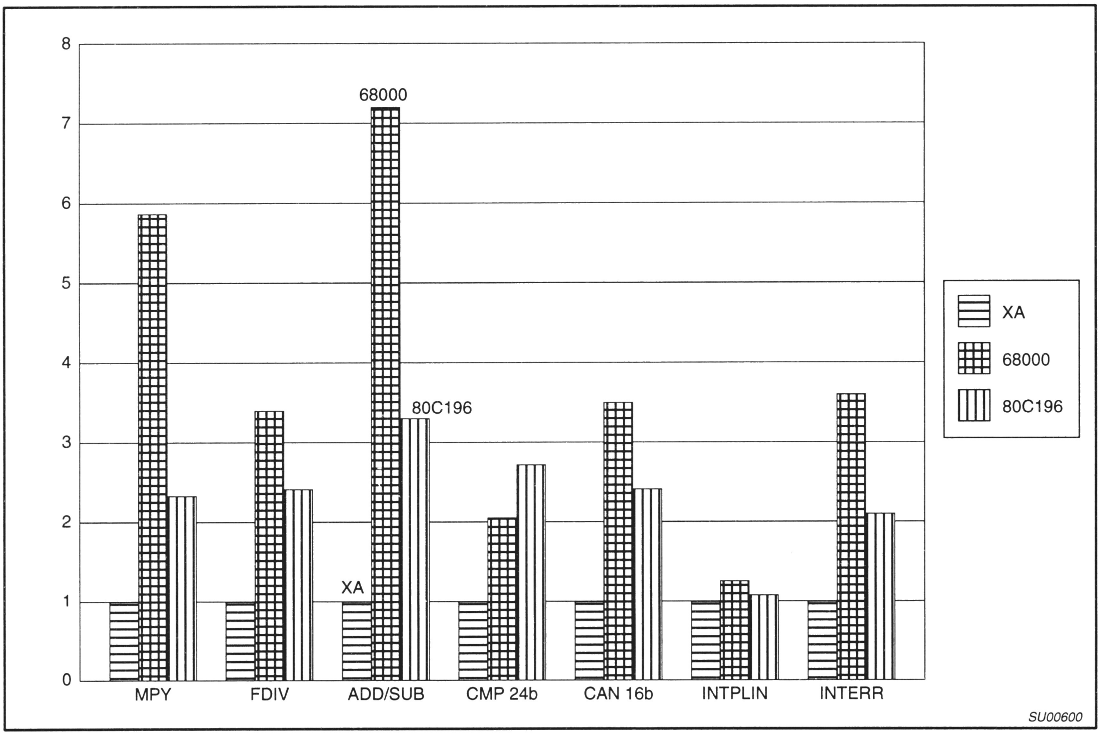
	XA	68000	80C196	8051
MPY	1	1	1.5	1
FDIV	1	0.89	1.06	2.6/1.11
ADD/SUB	1	3	1.75	2.5
CMP 24b	1	0.88	0.88	1
CAN 16b	1	1.75	0.75	1.5
INTPLIN	1	0.33	0.43	0.33
INTERR	1	2.24	1.41	1.71



XA EXECUTION TIME RESULTS

Graph showing performance with respect to 68000, and 80C196 cores normalized with respect to XA. The 80C51 is included just for reference.

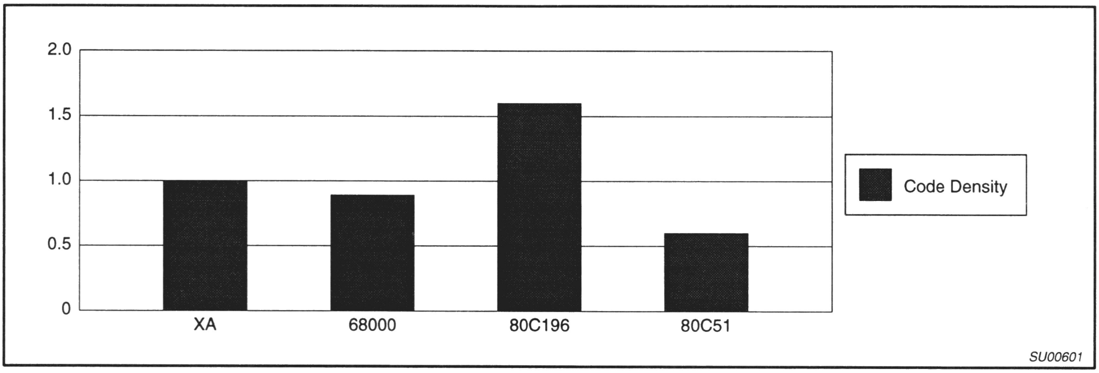
	XA	68000	80C196	8051
MPY	1	5.87	2.33	50
FDIV	1	3.4	2.41	86
ADD/SUB	1	7.2	3.3	19.74
CMP 24b	1	2.05	2.72	6.4
CAN 16b	1	3.5	2.41	11.54
INTPLIN	1	1.26	1.08	4.34
INTERR	1	3.6	2.1	5.16



BIT TEST BENCHMARK: CODE DENSITY NORMALIZED WITH XA (=1.0)

The 80C51 is shown here only for reference.

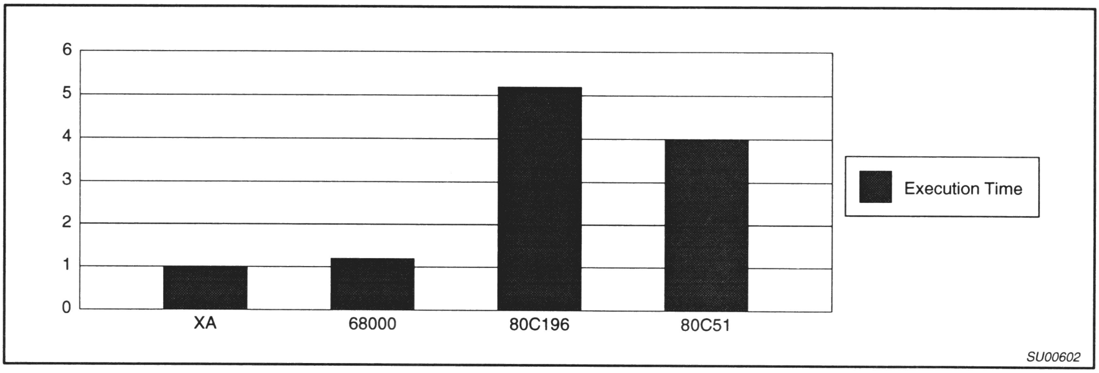
	XA	68000	80C196	8051
Code Density	1	0.89	1.6	0.6



BIT TEST BENCHMARK: EXECUTION TIME NORMALIZED WITH XA (=1.0)

The 80C51 is shown here only for reference.

	XA	68000	80C196	8051
Execution Time	1	1.2	5.2	4



An upward migration path for the 80C51: the Philips XA architecture

AN704

Author: *Greg Goodhue*

The 80C51 is arguably the most used 8-bit microcontroller architecture in the world, and a vast amount of public and private code exists for this processor. The "XA" (Extended Architecture) microcontroller, developed by Philips Semiconductors, is a high performance 16-bit processor that retains source code compatibility with the original 80C51. By permitting simple translation of source code, the XA allows existing 80C51 code to be re-used with a higher performance 16-bit controller. This provides an upward mobility path to a 16-bit controller for 80C51 users that has not previously existed, while also bringing a low cost, high performance, general purpose 16-bit controller to the market. How can a modern 16-bit controller provide compatibility with the venerable 80C51 without badly compromising the architecture and performance?

DESIGN TRADEOFFS

Many tradeoffs must be made and considerations taken into account when creating an upward compatible processor that must also be high performance and low cost. Among the areas to be considered are the processor's memory map and means of accessing memory, instruction set and methods of instruction execution, stack operation, interrupts, and special features added to enhance particular functions, such as multi-tasking, exception handling, and debugging features.

The goal of source code compatibility, rather than object code compatibility, was adopted for a number of reasons. First, absolute upward compatibility with an existing processor is by definition impossible if one of the goals of the new processor is to generally improve performance. By doing the same things in less time, the time related attributes of previously written code change.

Another consideration has to do with the fact that the 80C51 used all but one of the 256 opcodes available with an 8-bit opcode field. Adding more than a few new instructions or a new data type (such as 16-bit operations) would result in a very inefficient instruction encoding, and inefficient execution as well, for those new functions.

Creating a new instruction set that includes an exact copy of the 80C51 instruction set as a subset would also be very inefficient, since some subset of many new operations would act as duplicates of 80C51 instructions. For instance, a more powerful ADD instruction that can add any byte or word register to any other register is a superset of the 80C51 instruction to add a register to the accumulator. In such a case, there is no good argument to duplicate the original instruction precisely.

An 80C51 "mode" on an otherwise totally new (and therefore incompatible) processor was also considered. However, this approach would result in having in effect 2 processors on one chip, which would be confusing and not very cost effective. Mixing new, more efficient code with existing 80C51 code would require switching modes often, which would be very cumbersome and potentially hazardous. If a mode switch was skipped by accident in some seldom executed code sequence, the processor could suddenly find itself executing code using the wrong instruction set!

HOW IS IT DONE?

The team that created the XA architecture at Philips followed several rules in order to insure that 80C51 compatibility goals were met. First, translation for all (or nearly all) 80C51 instructions would be one to one. Multi-instruction combinations that could result in problems if split by an interrupt or otherwise compromise the integrity of the translation would be avoided. This has the effect of producing a simple, straightforward, and easily checkable translation.

Second, most 80C51 instructions should be a subset of new XA instructions. If that is not possible or doesn't make sense in a particular case, the original 80C51 instruction would be included "as-is", even though it might not fit the basic XA architecture's philosophy.

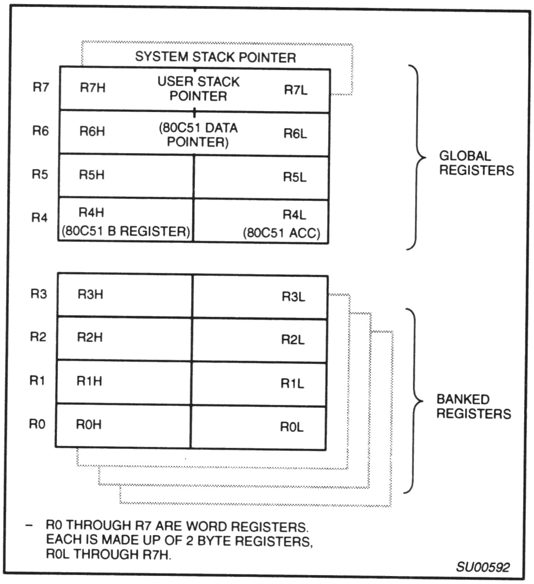
Third, XA register, code memory, data memory, and Special Function Register addressing would be a superset of the 80C51 equivalents. The same idea applies to other features that are part of the CPU.

Finally, some compromises to these compatibility rules are allowed in cases where keeping absolute compatibility would adversely affect system cost, high level language support, or performance. The cost (in engineering time) of dealing with any incompatibilities must be kept to a minimum. Preferably, the issue should not even be noticeable to most customers.

MEMORY MAPPING

At the root of any potential compatibility between the XA and the 80C51 is the memory map. The XA takes a simple but effective approach to this issue: its memory map is a superset of the 80C51 memory map. Modes of addressing memory likewise duplicate the modes available on the 80C51, adds new modes, and enhances some of the old ones.

In translating 80C51 source code to the XA, particular registers are used to represent the accumulator (A) and the data pointer (DPTR). Although the XA can use any of the 14 general purpose byte registers int he register file as an accumulator, the 80C51 has some features that require the accumulator to be a specific byte register. These are primarily the parity flag and a few special instructions that intrinsically reference the accumulator in a way that could not be generalized in the XA. The latter are, specifically, instructions like: JZ, JNZ, MOVC A,@A+DPTR, MOVC @A+PC, and JMP @A+DPTR. Figure 1 shows the register file of the first XA derivative (the XA architecture can support some additional registers not implemented in the first part) and the registers used for 80C51 translation.

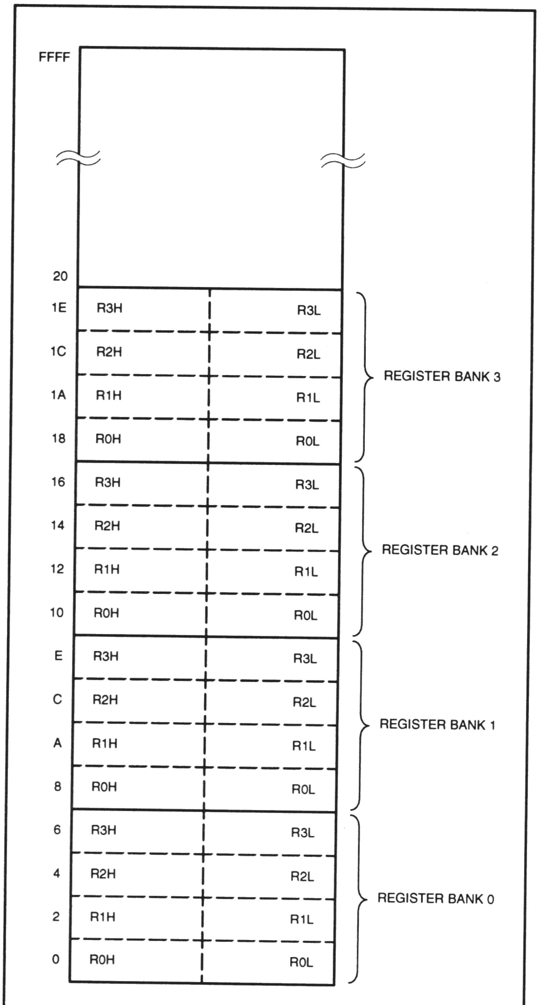

SYSTEM STACK POINTER
R7
R7H
USER STACK POINTER
R7L
R6
R6H
(80C51 DATA POINTER)
R6L
R5
R5H
R5L
R4
R4H (80C51 B REGISTER)
R4L (80C51 ACC)
GLOBAL REGISTERS
R3
R3H
R3L
R2
R2H
R2L
R1
R1H
R1L
R0
R0H
R0L
BANKED REGISTERS
SU00592

- R0 THROUGH R7 ARE WORD REGISTERS. EACH IS MADE UP OF 2 BYTE REGISTERS, R0L THROUGH R7H.

Figure 1. XA Register File

An alternate program status word (PSW) was created on the XA to duplicate the 80C51 PSW and contains the P (parity) flag as well as the F1 and F0 user defined flags that are not found in the native XA PSW. The XA PSW, on the other hand, adds some new status flags and system controls to expand its capabilities.

The XA register file duplicates the 4 banks of 8 bytes that are found in the 80C51. An 80C51 compatibility mode determines whether these locations appear both as registers and as the lower 32 bytes of data memory as they do on the 80C51. The more standard scheme of keeping the register file separate from the data memory is the default on the XA. Besides being "cleaner", the separation of the register file from data memory allows for a higher performance implementation of the XA processor core at some point in the future if and when 80C51 compatibility is no longer required. Figure 2 shows the overlap of data memory and the register file in compatibility mode. This shows only this one aspect of the XA memory map, not a general view of the memory.


FFFF
20
1E
R3H
R3L
1C
R2H
R2L
1A
R1H
R1L
18
R0H
R0L
REGISTER BANK 3
16
R3H
R3L
14
R2H
R2L
12
R1H
R1L
10
R0H
R0L
REGISTER BANK 2
E
R3H
R3L
C
R2H
R2L
A
R1H
R1L
8
R0H
R0L
REGISTER BANK 1
6
R3H
R3L
4
R2H
R2L
2
R1H
R1L
0
R0H
R0L
REGISTER BANK 0

NOTES:

1) The addresses shown are the data memory addresses that correspond to the register appearance in the XA data memory when 80C51 compatibility mode is activated.

2) This drawing represents a single XA data memory segment.

SU00593

Figure 2. XA Register File and Data Memory Overlap

A second aspect of XA memory addressing is also controlled by the aforementioned 80C51 compatibility mode. In the XA, indirect memory accesses normally make use of a 16-bit pointer register, which may be any of the word registers in the register file. The 80C51, however, allows only the 2 single-byte registers R0 and R1 to used for indirect references. The XA is forced use the first 2 single-byte registers in the currently selected bank as byte pointers rather than word pointers when the 80C51 compatibility mode is activated. Thus, translated 80C51 code typically must be run with the compatibility mode activated.

The data memory map for a single XA data segment looks just like the entire data memory map for an 80C51. This leads to the possibility of using a single XA to perform the function of several 80C51s, with a separate data segment and code area allocated to a task that was originally performed by one 80C51. The XA includes hardware support for multi-tasking operation in order to allow for this and other interesting possibilities.

The XA retains the direct and indirect addressing modes of the 80C51, although both are greatly expanded in capability, as shown in figure 3. The direct data addressing has been increased to use up to 1K bytes of data memory. Indirect addressing is done in 64K byte segments, for a total of up to 16 megabytes. Both types of addressing seamlessly switch from internal to external data memory wherever the boundary exists between the two for a particular chip. In this manner, the processor stack may also be extended off-chip up to nearly 64K bytes if necessary. Because of the seamless internal to external memory transition, the XA would not normally attempt off-chip data accesses at the low memory addresses that correspond to the on-chip data RAM. For that reason, the 80C51 MOVX instruction is included on the XA in order to allow translated code to run without changes in the external memory address map. This works because MOVX always forces data to be read from off-chip memory.

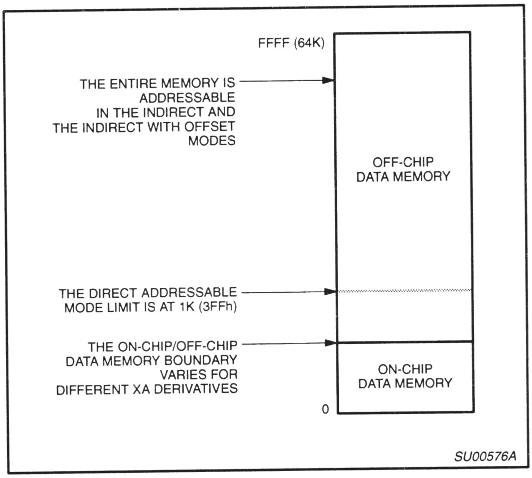

FFFF (64K)
THE ENTIRE MEMORY IS ADDRESSABLE IN THE INDIRECT AND THE INDIRECT WITH OFFSET MODES
OFF-CHIP DATA MEMORY
THE DIRECT ADDRESSABLE MODE LIMIT IS AT 1K (3FFh)
THE ON-CHIP/OFF-CHIP DATA MEMORY BOUNDARY VARIES FOR DIFFERENT XA DERIVATIVES
ON-CHIP DATA MEMORY
0
SU00576A

Figure 3. XA Memory Addressing

On the 80C51, the special function registers (SFRs) were mapped into the direct address space starting at location 128, through the end of that space at location 255. Since the 80C51 only allowed SFR access by direct addresses, where the entire address is encoded into the instruction, the XA does not need to duplicate its SFRs in exactly the same area or at the same specific addresses. In order to simplify the memory map, expand the SFR space, and expand the directly addressed data space, the XA defines a totally separate SFR space that is not logically related to the rest of data memory. To translate 80C51 source code, the original SFR name is kept in the translated code, unless the name was changed for some reason. In any case, as long as the reference is by name, a code translator need not try to determine which SFR it is, or where it belongs on a particular XA derivative. If 80C51 source code for some reason references an SFR by its address, a code translator might attempt to look it up in an SFR map for the 80C51 derivative to which the code was targeted.

A second mode control in the XA applies to 80C51 translated code, although it may be used in pure XA applications as well. This is the Page Zero, or PZ, mode. This mode forces the XA to only allow 64K of address space in both the data and code memories. The purpose is to reduce the overhead required to support the extra address space if it is not needed, such as in "single-chip" systems that do not use any off-chip data or program. Besides saving stack space for 24-bit subroutine and interrupt return addresses (reduced to 16 bits in PZ mode), overall XA operation is faster by having smaller stack pushes and pops. Since the 80C51 supported only 64K of code and data space, translated 80C51 code will likely fit into the same category.

There are other changes in the processor stack on the XA, besides the need to save 24 bits of return address when not running in the Page Zero mode. First, a great deal of extra hardware in the processor would be required to allow both byte and word pushes and pops on the stack, especially since word operations could then sometimes be mis-aligned from word address boundaries in the data memory, so stack operations on the XA are always done in word increments. Mis-aligned word operations, aside from being difficult to implement, would be very inefficient, since they would have to be split up into multiple byte operations. This means that translated 80C51 code run on the XA will tend to use somewhat more stack space than it did originally. The automatic save of the PSW during interrupts on the XA might also increase stack usage in some cases, since a few 80C51 programs may have been able to omit saving the PSW during interrupt processing.

Secondly, the XA stack has been altered so that the direction of growth is downward, conforming to the industry standard for stack operation on 16-bit processors. There is also a necessary relationship between the stack growth direction and the order in which the bytes of a word are stored in memory for a processor that is capable of stack relative addressing, as can be done with the XA. This relationship required that the stack grow downward since data on the XA is stored in memory with the low order byte of a word at the lower address (sometimes referred to as Little Endian storage order).

These differences in stack operation may require some changes to be made by the user for any 80C51 source code translated to the XA. In most cases, the change would be limited to choosing a different starting address for the stack.

A look at interrupt processing presents some other issues for 80C51 compatibility. In order to allow more powerful handling of interrupts, the XA has to make some compromises. Besides the previously mentioned fact that the PSW is automatically saved on the stack, which would have been done explicitly in 80C51 interrupt service code, the return address on the stack is also different if Page Zero mode is not active. So, any code written for the 80C51 which relied in some manner on manipulating the return address on the stack, or on the PSW not being saved and restored automatically, will require modification. Both of these situations should be very rare. The standard (non-Page Zero mode) XA interrupt stack frame is shown in Figure 4.

CPU FEATURES

Another difference in interrupt processing is that the XA uses a more efficient and flexible vector table for interrupts and exceptions instead of the fixed vector scheme of the 80C51. The vector table must reside at the bottom of the code memory, since this is the only region that is guaranteed to always exist in a system that uses on-chip ROM or EPROM for the program. Thus, during 80C51 code translation, code found at the 80C51 interrupt service locations must be moved to another location. Of course, an interrupt vector table must be added to any translated 80C51 program that makes use of interrupts, and a reset vector entry must be created for all XA programs.

A major enhancement to the XA is the addition of a general purpose interrupt priority scheme that can support up to 15 levels, compared to only 2 on standard 80C51 parts, and up to 4 on enhanced parts. This addition, however, requires some changes in the way interrupt priorities are handled. Two-priority interrupt systems on 80C51 derivatives used a single bit in a priority register to select the two levels. Four-priority systems extended this to two bits, but in 2 different registers for each interrupt source. Extending that approach to 15 levels would entail 4 bits in 4 different registers for each interrupt source, which is getting a bit ridiculous. For the XA, a more reasonable approach was taken: 4 bits in a single register control the priority of each interrupt source. Priorities for 2 separate interrupts are contained in each 8-bit priority register.

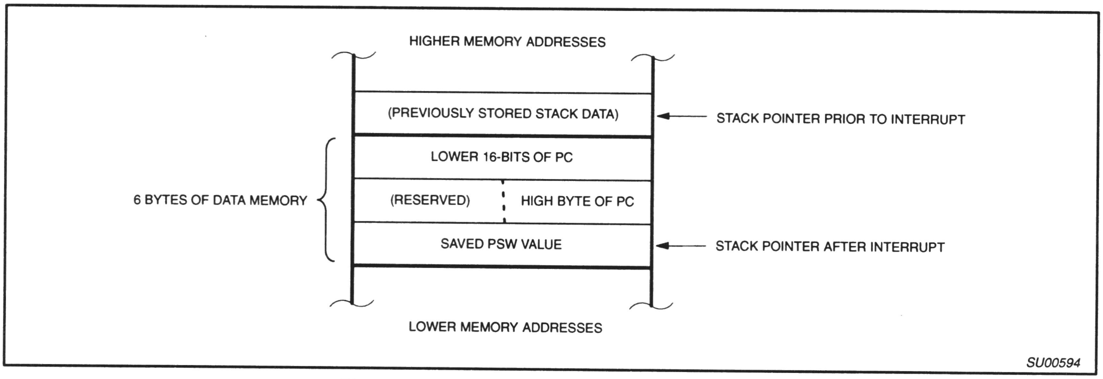

HIGHER MEMORY ADDRESSES
(PREVIOUSLY STORED STACK DATA)
STACK POINTER PRIOR TO INTERRUPT
LOWER 16-BITS OF PC
6 BYTES OF DATA MEMORY
(RESERVED)
HIGH BYTE OF PC
SAVED PSW VALUE
STACK POINTER AFTER INTERRUPT
LOWER MEMORY ADDRESSES
SU00594

Figure 4. Standard Interrupt Stack Frame on the XA

PERIPHERALS, ON AND OFF-CHIP

Another subject to look at is hardware compatibility. While complete hardware compatibility with the 80C51 was not a primary goal during the XA architecture development, hardware compatibility was retained whenever possible and practical. This particularly concerns peripheral devices such as UARTs, Timers, etc., and the processor's external bus system.

In the case of peripherals that are the same as those customarily found on the 80C51, these have been made to function as close as possible to the original, with some transparent enhancements such as framing error detection, overrun detection, and break detection in the UARTs. One exception to this general compatibility is that timer mode 0 of the standard timers 0 and 1, which is the rarely used 8048 compatible timer mode, has been replaced with a much more useful 16-bit auto-reload mode. In the future, further enhanced peripheral functions will likely lead eventually to completely new implementations that are not backward compatible with the 80C51.

Since there is no supposed relationship between the original oscillator frequency of an 80C51 system and a similar XA system using translated code, the exact relationship of peripheral speeds to the oscillator need not be preserved. For more flexibility in timer rates and therefore UART baud rates, the XA timers and some other peripherals are operated from a special clock whose rate is user programmable. The choices are the CPU clock divided by 4, 16, or 64, giving a wide range of uses. This function, like anything else in an application that is time critical, will need to be visited by the user when translated 80C51 code is used to drive XA peripherals.

The standard XA external bus interface includes all of the familiar 80C51 bus signals: ALE, $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{EA}}$, the multiplexed address and data bus, and address-only lines. However, some additional signals have been added and changes have been made in some of the details. For instance, the XA supports both 8-bit and 16-bit bus widths, using a second write signal to distinguish byte writes on a 16-bit bus. A WAIT line allows external circuitry to insert wait states into bus cycles for slow peripherals or program memories.

The largest change in the XA bus from the 80C51 is in the mapping of the multiplexed address and data lines. The 80C51 has a somewhat inefficient mapping that requires an ALE (Address Latch Enable) cycle in order to latch the least significant bits of an address for all external bus cycles. This was not a concern for the 80C51 due to its machine cycle timing, which allowed plenty of time for an ALE pulse. For the XA, which has no extra cycles during instruction execution, any extra strobes required on the bus during code fetches will likely take away time that could be used to execute instructions. As a result, the XA drives the 4 lower address lines directly, and does not require them to be latched. This means that the XA can fetch as many as 16 bytes of code between ALE cycles. The multiplexed address and data bus begins with the fifth address line (A4), paired with the first data line (D0), and continues to the width of the bus, either 8 or 16 bits. Above that will be more always-driven address lines, if more are needed by the application. Since the XA allows programming the number of address lines, those above the multiplexed portion of the bus need not be driven by the XA if they are not needed, leaving them free for other functions.

These changes mean that an XA device may be made pin compatible with a similar 80C51 derivative if the external bus is not used. Small changes to the external hardware must be made if the external bus is in use. Internally programmable bus cycle timing control on the XA allows programming the duration of all of the bus cycles, allowing nearly all memory and peripheral devices to be used on the XA bus without the need for an external WAIT state generator or any other additional circuitry.

INSTRUCTIONS REVISITED

The earlier mentioned goal of the XA to map nearly every 80C51 instruction to a single XA instruction was met. Just one 80C51 instruction cannot be replaced by single XA instruction. That instruction is XCHD (exchange digit), a seldom used 80C51 instruction. This unusual instruction exchanges the lower nibble of the 80C51 accumulator with a nibble at an internal RAM address pointed to by byte register R0 or R1. The XA would have required additional special circuitry in order to support this operation. As a result, it was decided to allow a multi-instruction sequence in this case, since the instruction is rarely used. The sequence used to replace XCHD is:

```
PUSH  R4H        ; save temporary register.
MOV   R4H, (Ri)  ; get second operand.
RR    R4H, #4    ; swap one byte.
RR    R4L, #4    ; swap second byte (the “A” register).
RL    R4, #4     ; swap word, result is swapped nibbles in A
                   and R4H.
MOV   (Ri), R4H  ; store result.
POP   R4H        ; restore temporary register.
```

Some additional code may be needed if an application requires this sequence to be un-interruptable for some reason. All other 80C51 instructions translate one-to-one to XA instructions. Since the XA instruction set and memory model are a superset of the 80C51, and since most mnemonics and names were kept the same, 80C51 code translated for the XA looks nearly the same as the original. Some examples are shown below.

Table 1. Examples of 80C51 to XA Source Code Translation

TYPE OF OPERATION	80C51 SOURCE CODE		XA SOURCE CODE	
Move immediate to SFR.	MOV	TCON,#00h	MOV.B	TCON,#00h
Move direct address to accumulator.	MOV	A,TstDat	MOV.B	R4L,TstDat
Move register to register.	MOV	R5,A		
Arithmetic with 2 registers.	ADD	A,R1	ADD.B	R4L,R0H
Arithmetic with register and immediate.	SUBB	A,#'0'	SUBB.B	R4L,#'0'
Increment a register.	INC	R0	ADDS.B	R0L,#1
Test a register.	CJNE	A,#'0',Cmd1	CJNE.B	R4L,#'0',Cmd1
Clear a bit.	CLR	RxFlag	CLR	RxFlag
Set a bit.	SETB	EX1	SETB	EX1
Test a bit.	JNB	RcvRdy,Wait	JNB	RcvRdy,Wait
Subroutine call.	ACALL	Test	CALL	Test
Subroutine return	RET		RET	
Push register onto stack.	PUSH	ACC	PUSH.B	R4L
Pop register from stack.	POP	ACC	POP.B	R4L

Details of instruction translation for the entire 80C51 instruction set are available in the Philips XA User Guide.

One side effect of source code compatibility of the XA with the 80C51 is that the number of bytes required to encode some instructions changes between the two processors. In most cases, this is not a major concern, however it does raise issues with the translated code for some situations. A simple example of this is that a conditional branch could have the target address move out of range when translated code is re-assembled. This should be a rare occurrence since the range of short relative branches on the XA has been doubled to 256 bytes forward or backward. The same issue does not exist for farther jumps and calls since the XA extends that range to beyond the entire 80C51 address range.

The precise length of a branch instruction is of concern in certain cases, such as a table of jump instructions entered using the JMP @A+DPTR instruction of the 80C51. The XA instruction set includes this jump, but does not include a 2-byte replacement for the 80C51 AJMP instruction which is often used in jump tables. The user will have to make small changes to the indexing into such a table if it is translated to run on the XA.

A similar issue can arise for a translation of the 80C51 instruction MOVC A,@A+PC, since the distance from this instruction to the lookup table that it is accessing may change. The solution is the same as for JMP @A+DPTR: some user intervention to adjust the table index.

User intervention will also be needed in any case where the timing of instructions in the original 80C51 code is of importance. The XA reduces the execution time of each instruction to the minimum possible with its internal hardware implementation. Also, instructions are normally fetched into a small queue prior to being needed to continue execution, which can lend additional uncertainty to execution times. The execution time of loops or the time between particular instructions can be calculated and adjusted by the use of NOPs, delay loops, or other means of matching timing. Also, any variable execution timing of the same code due to it being entered in different ways can be handled with certain coding techniques. An example would be a loop that is entered by "falling through" the preceding code on the first instance and branching back to be repeated on subsequent occasions. The branch back takes extra time not seen on the first entrance to the code due to the necessity of "flushing" the queue on a branch. The solution in this case is to add a branch instruction prior to the loop branching to the first instruction of the loop. Then, each cycle through the loop acquires the same timing. Of course, a simple source code translator cannot sense such cases and attempt to deal with them automatically.

AN EXAMPLE

As an example of translating 80C51 source code into XA source code, an actual piece of 80C51 code from a working application was taken and translated using the rules that were presented above. The results of the simple one-to-one translation are shown below.

Table 2. Sample 80C51 Routines Translated for the XA

Original 80C51 source code:

```
; Sets up UART and Timer (for baud rate generation), prints a string,
; and prints a hexadecimal value.

Start:    MOV     SCON,#42h       ; Set UART for 8-bit variable rate.
          MOV     TMOD,#20h       ; Set Timer1 for 8-bit auto-reload.
          MOV     TCON,#00h       ; Stop timer 1 and clear flag.
          MOV     TL1,#0FDh       ; Set timer for 9600 baud @ 11.0592 MHz.
          MOV     TH1,#0FDh       ; Set reload register for same rate.
          MOV     A,PCON          ; Make sure SMOD bit in PCON is
          CLR     ACC.7           ;   cleared for this baud rate.
          MOV     PCON,A
          SETB    TR1             ; Start timer

          MOV     DPTR,#Msg1      ; Send a stored message.
          ACALL   Msg

          MOV     A,P1            ; Send Port 1 value as hexadecimal.
          ACALL   PrByte
           .       .
           .       .
           .       .

;*************************************************
;          Subroutines
;*************************************************

; Print byte routine: print ACC contents as ASCII
;   hexadecimal.

PrByte:   PUSH    ACC
          SWAP    A
          ACALL   HexAsc
          ACALL   XmtByte
          POP     ACC
          ACALL   HexAsc          ; Print nibble in ACC as ASCII hex.
          ACALL   XmtByte
          RET

; Hexadecimal to ASCII conversion routine.
; Converts a nibble to ASCII hex.

HexAsc:   ANL     A,#0FH
          JNB     ACC.3,NoAdj
          JB      ACC.2,Adj
          JNB     ACC.1,NoAdj
Adj:      ADD     A,#07H
NoAdj:    ADD     A,#30H
          RET
```

Translated XA source code:

```
Start:    MOV.B   SCON,#42h
          MOV.B   TMOD,#20h
          MOV.B   TCON,#00h
          MOV.B   TL1,#0FDh
          MOV.B   TH1,#0FDh
          MOV.B   R4L,PCON
          CLR     R4L.7
          MOV.B   PCON,R4L
          SETB    TR1

          MOV.W   R6,#Msg1
          CALL    Msg

          MOV.B   R4L,P1
          CALL    PrByte
           .       .
           .       .
           .       .

PrByte:   PUSH.B  ACC
          RL.B    R4L,#4
          CALL    HexAsc
          CALL    XmtByte
          POP.B   ACC
          CALL    HexAsc
          CALL    XmtByte
          RET

HexAsc:   AND.B   R4L,#0FH
          JNB     R4L.3,NoAdj
          JB      R4L.2,Adj
          JNB     R4L.1,NoAdj
Adj:      ADD.B   R4L,#07H
NoAdj:    ADD.B   R4L,#30H
          RET
```

Original 80C51 source code:

```
; Message string transmit routine.

Msg:      PUSH    ACC
          MOV     R0,#0          ; R0 is character pointer (string
MsgL:     MOV     A,R0           ;   length is limited to 256 bytes).
          MOVC    A,@A+DPTR      ; Get byte to send.
          CJNE    A,#0,Send      ; End of string is indicated by a 0.
          POP     ACC
          RET

Send:     ACALL   XmtByte        ; Send a character.
          INC     R0             ; Next character.
          SJMP    MsgL

Msg1:     DB      0Dh,0Ah,0Dh,0Ah
          DB      'Port 1 value = ', 0

; Wait for UART ready, then send a byte.

XmtByte:  JNB     TI,$
          CLR     TI
          MOV     SBUF,A
          RET
```

Translated XA source code:

```
Msg:      PUSH.B  ACC
          MOV.B   R0L,#0
MsgL:     MOV.B   R4L,R0L
          MOVC.B  A,[A+DPTR]
          CJNE.B  R4L,#0,Send
          POP.B   ACC
          RET

Send:     CALL    XmtByte
          ADDS.B  R0L,#1
          BR      MsgL

Msg1:     DB      0Dh,0Ah,0Dh,0Ah
          DB      'Port 1 value = ',0


XmtByte:  JNB     TI,$
          CLR     TI
          MOV.B   SBUF,R4L
          RET
```

The translated XA code looks very much like the 80C51 source code and can easily be read by anyone familiar with the original program. Statistics for this example are shown in the following table.

Table 3. Statistics on Sample 80C51 to XA Code Translation

STATISTIC	80C51 CODE	XA TRANSLATION	COMMENTS
Bytes to encode	107	151	– Includes NOPs added for branch alignment on XA.
Clocks to execute	840	212	– Raw execution time for instructions in code, without flow analysis. Conditional branch times calculated as if half taken, half not taken.
Time to execute @ 20 MHz	42 sec	10.6 sec	– A 4x speed improvement for a simple translation with no optimization.

SOME XA ENHANCEMENTS

The subject of this article has been how the new Philips XA microcontroller architecture supports upward compatibility with the 80C51. The XA adds quite a bit to the equation beyond mere 80C51 compatibility, which has barely been touched upon here. In addition to high performance and very compact instruction encoding, the XA is specifically designed for high level language support for compilers such as C, has many features to support multi-tasking, with protected features and separate memory spaces, many 32-bit operations in addition to general 16-bit arithmetic, and greatly enhanced interrupt processing, to name a few. A complete description of all of these features and many more may be found in the XA User Guide and data sheets for specific parts.

THE UPWARD SPIRAL

Many openings have been left in the XA architecture for even more enhancements in the future, such as full pipelining, complete 32-bit operation support, or a faster peripheral bus. The XA is the foundation of a new microcontroller derivative family in a manner similar to the very popular 80C51 family. Many other advanced microcontroller architectures have been brought to market since the 80C51 was designed years ago. But until now, none has allowed the enormous quantities of 80C51 code that users have on file to be re-used with minimal effort on a state-of-the-art 16-bit processor. With the Philips XA, that is now possible, while getting the benefit of a modern 16-bit processor with few compromises.

Section 6

Development Support Tools

CONTENTS

XA tools linecard 478

Ashling: CTS51 Microprocessor development systems for Philips microcontrollers 479

CEIBO: DB-XA Development Board 484

EDI Corporation: Accessories for 8051-Architecture Devices 486

Emulation Technology, Inc.: XA Microcontroller Development Tools 487

Hi-Tech C Compiler for the Philips XA microcontroller – technical specifications 489

Logical Systems: Adapters for Philips 51XA-G3 492

Nohau: EMUL 51XA In-Circuit Emulator for the Philips 80C51XA 493

Philips Semiconductors / Macraigor Systems: 80C51XA Software Development Tools . . 495

P51XA Development Board/Emulator 497

Signum Systems: Universal In-Circuit Emulator for 8051/31 Series 499

System General: Universal Device Programmers 501

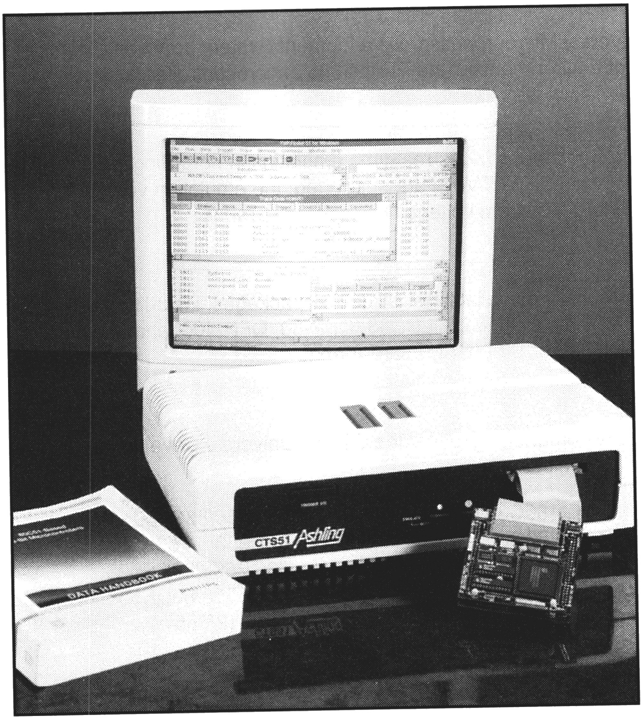
XA tools linecard

	Telephone/Contact					Product
	North America		Europe			
C Compilers						
Archimedes	1–206–822–6300	Mary Sorensen	SW	41.61.331.7151	Claude Vonlanthen	C–51XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	C–XA
Hi–Tech	1–207–236–6713	Avocet – T. Taylor	UK	44.1.932.829460	Computer Solutions	Hi–Tech C (XA)
Franklin Software	1–408–296–8051	Siegfried Bleher	US	1–408–296–8051	Siegfried Bleher	XA–CD (5050)
Sierra Systems	1–510–339–1976	Larry Rosenthal	US	1.510.339.1976	Larry Rosenthal	Sierra C (XA)
Emulators (including Debuggers)						
Ashling	1–508–366–3220	Bob Labadini	IR	353.61.336644	Micheal Healy	Ultra2000–XA
Cactus Logic	1–818–337–4547	Joel Lagerquist	US	1.818.337.4547	Joel Lagerquist	IDS
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	DS–XA
Emulation Tech	1–408–982–0660	Joseph J. Bagliere	US	1.408.982.0660	Joseph J. Bagliere	Various
Nohau	1–408–866–1820	Steve Ehert	SW	46.40.922425	Mikael Johnsson	EMUL51XA–PC
Cross Assemblers						
Archimedes	1–206–822–6300	Mary Sorensen	SW	41.61.331.7151	Claude Vonlanthen	A–51XA
Ashling	1–508–366–3220	Bob Labadini	IR	353.61.336644	Micheal Healy	SDS–XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	ASM–XA
Franklin Software	1–408–296–8051	Siegfried Bleher	US	1–408–296–8051	Siegfried Bleher	XA–ASM (4050)
Philips/Macraigor*	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	Mcgtool
Real–Time Operating Systems						
CMX Company	1–508–872–7675	Charles Behrmann	US	1.508.872.7675	Charles Behrmann	CMX–RTX RTOS
Embedded Sys Prods	1–713–728–9688	Ron Hodge	US	1.713.728.9688	Ron Hodge	RTXC
R&D Publications	1–913–841–1631	Customer Service	US	1.913.841.1631	Customer Service	Labrosse MCU/OS
Simulators						
Archimedes	1–206–822–6300	Mary Sorensen	SW	41.61.331.7151	Claude Vonlanthen	SimCASE–51XA
Avocet Systems	1–207–236–9055	Jamie Arrison	US	1.207.236.9055	Jamie Arrison	AvCase–51XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	DEBUG–XA
Franklin Software	1–408–296–8051	Siegfried Bleher	US	1–408–296–8051	Siegfried Bleher	XA–DK (8250)
Philips/Macraigor*	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	Mcgtool
Translators (80C51–to–XA)						
Ashling	1–508–366–3220	Bob Labadini	IR	353.61.336644	Micheal Healy	N.A. — FC–51XA Eur. — Ultra2000–XA
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	CONV–XA
Philips/Macraigor*	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	Mcgtool
Development Boards						
Ceibo	1–314–830–4084	Roy Schwartzman	GE	49.6151.27505	M. Kimron	DB–XA
Philips/Macraigor	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	P51XA–DBE SD
EPROM Programmers						
BP Microsystems	1–800–225–2102	Sales Department	US	1.713.688.4600	Sales Department	BP–1200
Ceibo	1–314–830–4084	Roly Schwartzman	GE	49.6151.27505	M. Kimron	MP–51
Data I/O Corp.	1–800–247–5700	Tech Help Desk	BE	32.1.638.0808	Roland Appeltants	UniSite
Philips/Macraigor	1–408–991–51XA	Mike Thompson	US	1.408.991.5192	Mike Thompson	P51XA–DBE SD
Adapters & Sockets						
EDI Corp	1–702–735–4997	Milos Krejcik	US	1.702.735.4997	Milos Krejcik	44PG/44PL
Logical Systems	1–315–478–0722	Lynn Burko	US	1.315.478.0722	Lynn Burko	PA–XG3FC–44

* The Philips cross assembler, simulator, and translator are available on the Philips BBS. Call 1–408–991–2406 or 31–40–721102. File name XA–TOOLS.EXE

Now supports XA!

CTS51 Microprocessor Development Systems for Philips Microcontrollers



Ashling's CTS51 Universal Microprocessor Development System for Philips Microcontrollers

Features of Ashling's Microprocessor Development systems:

- Integrated Development Environment for Philips 8051 under Windows
- Real-time in-circuit emulator for all Philips 8051 derivatives; full-speed, non-intrusive emulation
- Real-time, DSP-based Performance Analysis and Code Coverage systems for Software Quality Assurance
- Source-Level Debugging for 8051 programs, under Windows and DOS hosts
- Emulation and probe support for Philips low-voltage microcontrollers and Smart-Card microcontrollers
- Hardware break-before-execute breakpoints at every code and data address
- Stand-alone system, with built-in power supply; interchangeable probe cards for all derivatives and packages
- Programmer for all Philips 8051-family EPROM and EEPROM microcontrollers
- ISO9001-Certified Supplier; Ashling Microsystems Ltd. is certified to EN ISO9001/ASQC Q91

The Development Systems Company **Ashling**

Now supports XA!

Product Information

Ashling's CTS51 Universal Microprocessor Development System provides a complete hardware and software development environment for the Philips 8051 microcontroller family and all of its derivatives.

In-Circuit Emulator

The CTS51 In-Circuit Emulator provides real-time emulation in both Single-Chip and Expanded modes. Target voltages in the range 2.7 - 5 Volts are supported; the emulation voltage can be set at 3.0, 3.3V or 5.0V, or can track the target system voltage throughout the range.

ICE Probes

In-Circuit emulation probes are available for all device-packages, including DIP-24, DIP-28, DIP-40, PLCC-44, QFP-44, SDIP-42, SDIP-52, PLCC68, QFP-80, QFP-100, and Philips Smart-Card Microcontrollers in ISO7816 or SIM (GSM) card-format.

Devices Supported

Using field-upgradeable personality probes, the CTS51 Universal Development System supports all Philips 8051 derivatives, including:

80C51B/80C31B
83C552/80C552
83C750 83C751 83C752
83C592/80C592
83CL410 83CL580
80C31/80C52 80CL51
83C562/80C562
83C851
83C528/80C528 83C575
83CL782
83C550/83C550
83C053/54 83C654
83CE598/80CE598
83CE558/80CE558
83C852/855
80C51FA/FB/FC
PCD509x
SAA5290
83CL434
83CL168

Support for new Philips microcontrollers

Ashling's close technical collaboration with Philips Semiconductors ensures that new Philips microcontroller devices and device-architectures are supported at, or soon after, their introduction by Philips. New Ashling personality probes are regularly introduced for new standard microcontrollers, and microcontroller-based ASICs from Philips.

Philips Smart Card development support

Developed in co-operation with Philips, Ashling's CTS51 Microprocessor Development System provides a complete development-environment for Philips' 8CC852, 83C852 family of Smart Card Microcontrollers. Probes are available for ISO7816 and GSM (SIM) card formats. Ashling also supplies the SCPC4281 Smart Card Verification kit for stand-alone smart-card program emulation.

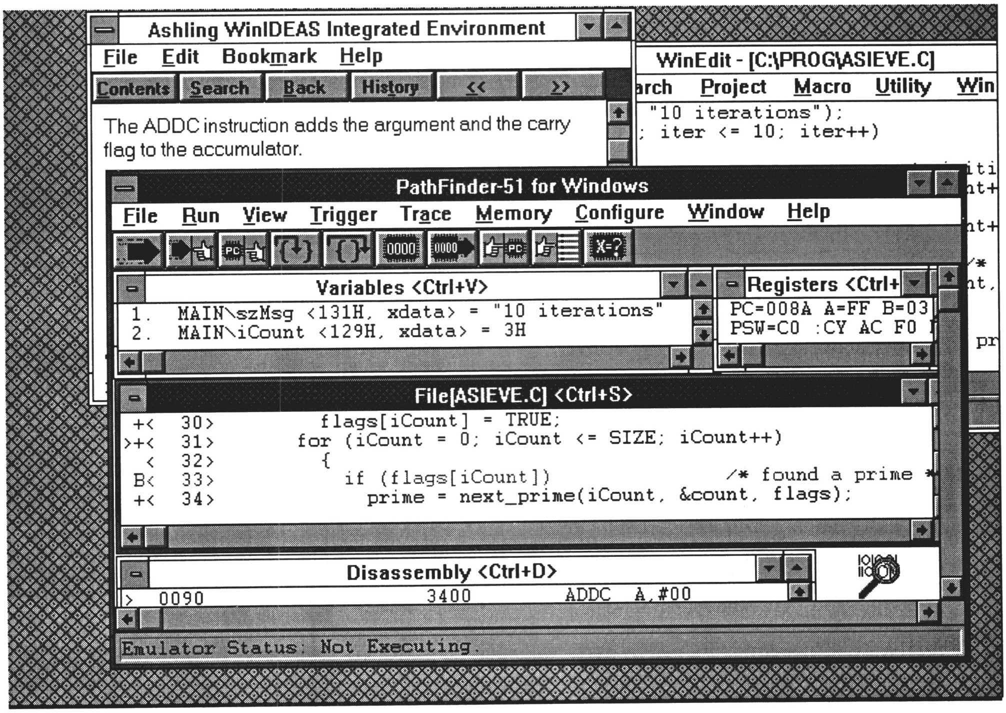
ISO9001 Certification; Quality Management System

Ashling Microsystems Ltd. operates a company-wide Quality Management System, formally certified to the ISO9001 international standard (NSAI Registration No. M619). This certification applies to all of Ashling's product development, manufacturing and customer-support activities.

NSAI
I.S. EN ISO 9001
Registration No.: M619

The Development Systems Company Ashling

Now supports XA!


Ashling WinIDEAS Integrated Environment
File Edit Bookmark Help
Contents Search Back History << >>
WinEdit - [C:\PROG\ASIEVE.C]
arch Project Macro Utility Win
PathFinder-51 for Windows
File Run View Trigger Trace Memory Configure Window Help
PC
0000
X=?
Variables <Ctrl+V>
Registers <Ctrl+
File[ASIEVE.C] <Ctrl+S>
Disassembly <Ctrl+D>

The ADDC instruction adds the argument and the carry flag to the accumulator.

```
 "10 iterations");
; iter <= 10; iter++)
```

```
1.  MAIN\szMsg <131H, xdata> = "10 iterations"
2.  MAIN\iCount <129H, xdata> = 3H
```

```
PC=008A A=FF B=03
PSW=C0 :CY AC F0
```

```
 +<  30>         flags[iCount] = TRUE;
>+<  31>       for (iCount = 0; iCount <= SIZE; iCount++)
  <  32>         {
 B<  33>           if (flags[iCount])                /* found a prime
 +<  34>             prime = next_prime(iCount, &count, flags);
```

```
>  0090                 3400          ADDC   A,#00
```

```
Emulator Status: Not Executing.
```

Ashling WinIDEAS: With the Windows Integrated Development Environment for Ashling Systems supports the full Philips Microcontroller family. A single keystroke moves you between the edit, help-lookup, compile, debug and emulate stages.

Integrated Development Environment

WinIDEAS, the Windows Integrated Development Environment for Ashling Systems, allows you to edit, compile, assemble, simulate, debug, download and execute code on your Philips microcontroller target system in the Windows environment throughout. WinIDEAS provides a uniform, flexible and extensible windows interface for Editing, C Compiling, Assembling, Fuzzy-logic design and compilation, Linking, In-circuit emulation, Performance Analysis, Code Coverage Measurement, Software Validation reporting, and EPROM/EEPROM programming on Philips Microcontroller projects.

System Execution Analyser

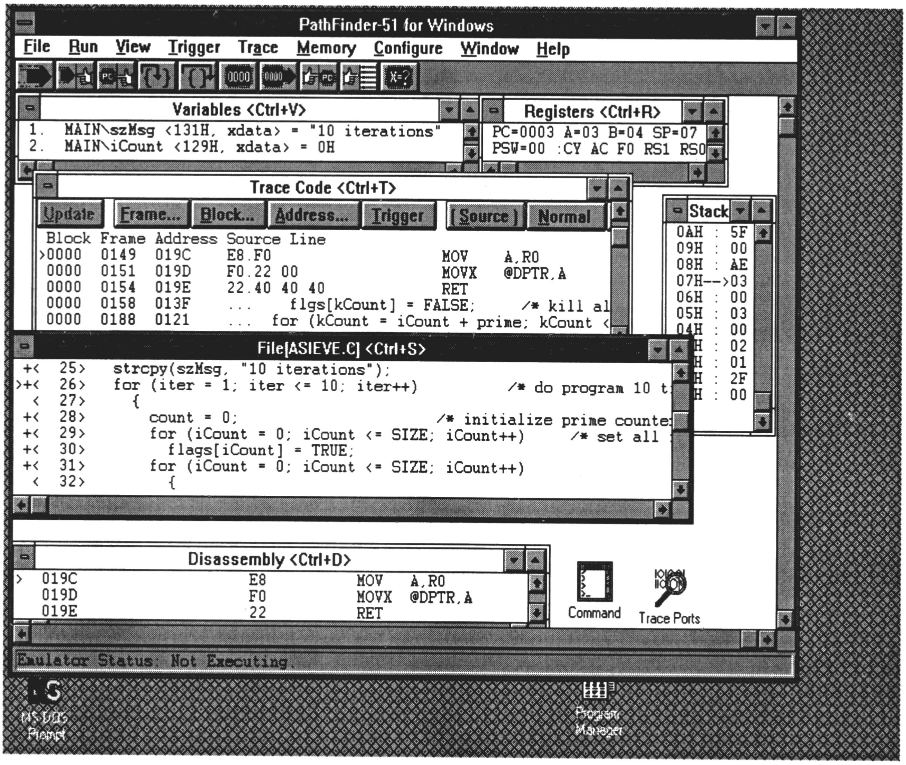
The System Execution Analyser (SEA) is a built-in, DSP-based option for Ashling's Universal Microprocessor Development System. It provides real-time, non-intrusive, non-statistical Performance Analysis, Code Coverage and Report Generation. The SEA also provides symbolic function-trace, time-stamping, timing analysis and software verification. You can measure maximum, minimum and average execution times, execution counts and percentage code execution at Source or Assembly level.

The Development Systems Company **Ashling**

Now supports XA!

Code Coverage System

The Code Test Management (CTMS) option for the Ashling's Universal Microcontroller Development System measures 8051 code execution in real-time throughout a test session. It maps tested and untested code; identifies all tested, partially-tested or untested C source statements and assembler instructions; generates formal, annotated, test reports; identifies redundant code; and provides a formal measurement of test completeness.


PathFinder-51 for Windows
File
Run
View
Trigger
Trace
Memory
Configure
Window
Help
Variables <Ctrl+V>
Registers <Ctrl+R>
Trace Code <Ctrl+T>
Update
Frame...
Block...
Address...
Trigger
[Source]
Normal
Stack
File[ASIEVE.C] <Ctrl+S>
Disassembly <Ctrl+D>
Command
Trace Ports
MS-DOS Prompt
Program Manager

```
1.  MAIN\szMsg <131H, xdata> = "10 iterations"
2.  MAIN\iCount <129H, xdata> = 0H
```

```
PC=0003 A=03 B=04 SP=07
PSW=00 :CY AC F0 RS1 RS0
```

```
 Block Frame Address Source Line
>0000  0149  019C     E8.F0                    MOV    A,R0
 0000  0151  019D     F0.22 00                 MOVX   @DPTR,A
 0000  0154  019E     22.40 40 40              RET
 0000  0158  013F     ...     flgs[kCount] = FALSE;      /* kill al
 0000  0188  0121     ...   for (kCount = iCount + prime; kCount <
```

```
0AH  :  5F
09H  :  00
08H  :  AE
07H-->03
06H  :  00
05H  :  03
04H  :  00
 H  :  02
 H  :  01
 H  :  2F
 H  :  00
```

```
 +<  25>   strcpy(szMsg, "10 iterations");
>+<  26>   for (iter = 1; iter <= 10; iter++)           /* do program 10 t
  <  27>     {
 +<  28>       count = 0;                           /* initialize prime counte
 +<  29>       for (iCount = 0; iCount <= SIZE; iCount++)      /* set all
 +<  30>         flags[iCount] = TRUE;
 +<  31>       for (iCount = 0; iCount <= SIZE; iCount++)
  <  32>         {
```

```
>  019C              E8        MOV    A,R0
   019D              F0        MOVX   @DPTR,A
   019E              22        RET
```

```
Emulator Status: Not Executing.
```

This PathFinder for Windows screen shows the Source program, Disassembly, Real-Time Trace (source-level, disassembly and opcodes), Variables-watch, On-chip Registers and Stack contents for an 8051 program.

Windows Source-Level Debugger

Ashling's "PathFinder for Windows" Source-Level debugger supports all popular microcontroller C and PL/M Compilers and Assemblers for the Philips microcontroller device family. PathFinder provides up to 20 display-windows, controlled by mouse, menu-bar, command-line, accelerator keys or button-bar. Assembly-language source level debugging is a unique feature of PathFinder for Windows.

The Development Systems Company Ashling

Now supports XA!

Microcontroller EPROM/EEPROM Programming

Ashling's CTP51 Prom Programming System is used with the Universal Microprocessor Development System, to program the entire Philips 8751 EPROM/EEPROM microcontroller family. Features include:

- Programming support for all Philips 8751-family EPROM microcontrollers, including 87C51, 87C51FC, 87C52, 87C528, 87C550, 87C552, 87C592, 87C751 and 87C752; plus all Philips 89C51-family EEPROM microcontrollers, including 89CE528, 89CE558, 89CE598 and SAA5290NV.
- Software-driven programming data files allow easy upgrade for new Philips devices.
- Programming Adapters are available for all Philips EPROM/EEPROM microcontroller packages.

Ashling Microsystems' Distributors:

AUSTRALIA:	Metromatics Pty. Ltd.	Tel: 07-3585155	Fax: 07-2541440
AUSTRIA:	Ashling Mikrosysteme	Tel: 08202 1276	Fax: 08202 8745
BELGIUM:	Air Parts Electronics	Tel: 02 241 64 60	Fax: 02 241 81 30
FRANCE:	Ashling Microsystèmes sarl	Tél: (1) 46.66.27.50	Fax: (1) 46.74.99.88
GERMANY:	Ashling Mikrosysteme	Tel: 08202 1276	Fax: 08202 8745
HUNGARY:	Vanguard Kft.	Tel: (1) 156-9000	Fax: (1) 156-8982
IRELAND:	Ashling Microsystems Ltd	Tel: 061-334466	Fax: 061-334477
ISRAEL:	RDT Ltd.	Tel: 03-6450745	Fax: 03-6478908
ITALY:	All-Data s.r.l.	Tel: 02-66015566	Fax: 02-66015577
KOREA:	DaSan Technology	Tel: (02) 501 8277	Fax: (02) 501 8276
NETHERLANDS:	Air Parts b.v.	Tel: 01720-43221	Fax: 01720-20651
SPAIN:	Sistemas Jasper s.l.	Tel: (1) 803 8526	Fax: (1) 803 8526
SWEDEN:	Ferner Elektronik AB	Tel: 08-760 8360	Fax: 08-760 8341
SWITZERLAND:	Litronic AG	Tel: 061 421 3201	Fax: 061 421 1802
U.K.:	Ashling Microsystems Ltd.	Tel: (01628) 773070	Fax: (01628) 773009
U.S.A:	Eastern Systems Inc	Tel: (508) 366 3220	Fax: (508) 366 1520

Ashling Microsystèmes sarl
2, rue Alexis de Tocqueville
Parc d'Activités Antony 2
92183 ANTONY - France.
Tél: (1) 46.66.27.50
Télécopieur: (1) 46.74.99.88

Ashling Mikrosysteme
Waldstraße 18
86510 Ried-Baindlkirch
Germany.
Tel: 08202-1276
Fax: 08202-8745

Ashling Microsystems Ltd.
Butler House, Market St.
Maidenhead
Berks. SL6 8AA, U.K
Tel: (01628) 773070
Fax: (01628) 773009

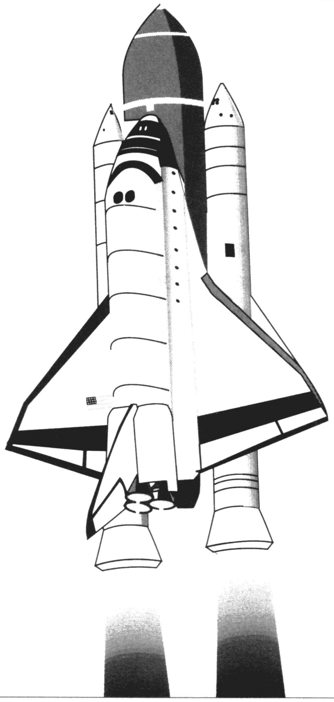
Ashling Microsystems Ltd
Plassey Technological Park
Limerick
Ireland
Tel: +353-61-334466
Fax: +353-61-334477

Eastern Systems Inc.
160 East Main Street
Westboro
MA 01581, USA
Tel: (508) 366 3220
Fax: (508) 366 1520

The Development Systems Company Ashling

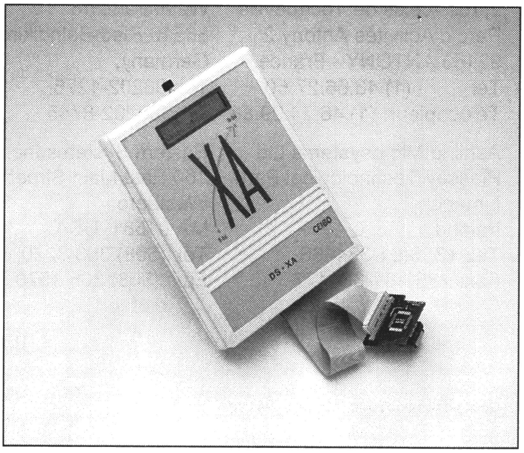
doc:philipdb.doc/cb Oct 94 ver. 1.0

CEIBO
DB-XA Development Board



Launching Tools for the New Philips 80C51XA Microcontroller Architecture

FEATURES
- Emulates XA microcontrollers and derivatives
- Real-time operation up to XA f_{MAX}
- Full simulator debugger
- Complete assembler and linker
- Source-level debugger for C and Assembler
- Runs under DOS and Windows software
- 256K of code memory expandable to 1M
- Data memory with mapping capabilities
- Performance analyzer
- Real-time and conditional breakpoints
- Emulation header and signal testpoints
- Serially linked to IBM PC at 115K baud



CEIBO DB-XA Development Board

DB-XA is a development board dedicated to all Philips XA microcontroller derivatives. It is serially linked to a PC/XT/AT or compatible system and can emulate the microcontroller using either the built-in clock oscillator or any other clock source connected to the microcontroller. The system emulates the microcontroller in ROMless mode. A two microcontroller architecture leaves the serial port for user applications. The software includes a source level Debugger for C and Assembler, on-line assembler and disassembler, software trace, conditional breakpoints and many other features. All the debugger functions run under DOS and Windows operating systems. The code memory permits downloading and modifying of user's programs. Mapping the data memory to a target circuit or to the system is possible. Breakpoints allow real time execution until an opcode is executed at a specified address or line of the source code. All I/O lines are easily accessed and may be connected to the on-board switches and LEDs to try a specific idea. The system is supplied with a User's Manual, software, emulation cable and a power supply.

Specifications

SYSTEM MEMORY

DB-XA provides 256KBytes of user code memory expandable to 1MByte. Memory may be configured in 8 or 16 bit modes. This RAM memory permits downloading and modifying of user's programs and variables.

CODE MEMORY

The system includes 256KBytes of RAM to be used as code memory, 4KBytes of this memory are reserved for system usage.

DATA MEMORY

Data memory can be mapped as belonging to the system or to the target circuitry.

BREAKPOINTS

Breakpoints allow real-time program execution until an opcode is executed at a specified address.

SOFTWARE TRACE

Program execution can be recorded in a 64K buffer. Conditional breakpoints may be defined to stop program execution. The user can define events and variables to be added to the software trace. The software trace is not a real-time function and is performed by slowing down the emulation speed.

PERSONALITY ADAPTERS

DB-XA uses standard microcontrollers for hardware and software emulation. The selection of a different microcontroller is made by replacing the microcontroller on the board by the appropriate microcontroller or daughter board.

SUPPORTED MICROCONTROLLERS

The supported microcontrollers are all of the Philips XA derivatives.

FREQUENCY

The system includes a crystal oscillator able to support different clock frequencies. The operating frequency range is from the microcontroller fmin to fmax.

USER SOFTWARE

DB-XA software works under two environments: DOS and Windows. The DOS is based on pull-down menus. The additional Windows program carries out the same functions as the DOS program with all the features and benefits of the new operating system.

DEBUG-XA

DEBUG-XA is a source level debugger for the XA architecture. The DEBUG-XA enables fast and reliable program debugging at source level for C-XA, ASM-XA and other compilers with standard object formats.

The DEBUG-XA can execute your code on a target emulator or as a simulator.

ASSEMBLER

The system is supplied with ASM-XA, a complete assembler and linker software package.

EMULATION RESTRICTIONS

The following emulation restrictions are valid for DB-XA:

1. RD and WR lines may not be used as I/O ports. There are no restrictions while RD and WR are activated by Move External Data instructions.
2. The system uses some of the microcontroller resources to emulate it: one interrupt according to software selection and 5 Bytes of the internal stack.

HOST CHARACTERISTICS

IBM PC/XT/AT or compatible systems with 640 KBytes of RAM, one floppy disk drive, one RS-232 interface card for the PC, PC-DOS 5.0 or later.

INPUT POWER

5VDC power supply.

MECHANICAL DIMENSIONS

20cm x 20cm

ITEMS SUPPLIED AS STANDARD

Emulation board, 40-pin emulation header, user software including source level debugger, simulator, assembler, User's Manual, RS-232 interface cable and power supply.

WARRANTY

Six months limited warranty, parts and labor.

For more information call CEIBO: 1-800-833-4084

Accessories for 8051-Architecture Devices

Device	App[1]	Description	Basic Adapter P/N[2]
8031AH, 8052AH, 80C32, 8051AH, 8XC51FA, 8XC51FB, 8XC51FC, 80C52, 87C52, 8XC504, 8XC524, 8XC575, 8XC576, 8XC652, 8XC654, & 80C851	E	40-DIP to 44-PLCC	40D/44PL-8051
	E	40-DIP to 44-QFP, 2-piece surface mount	40D/44QF31-TOP-8051 with 44QF31-SD
	E	44-PGA to 44-QFP, 2-piece surface mount	44PG/44QF31-TOP with 44QF31-SD
	E	44-PLCC to 44-QFP, 2-piece surface mount	44PL/44QF31-TOP with 44QF31-SD
80C39, 80C49, 8XC54, 8XC528, 8XC852, 8XCL410	E	44-PGA to 44-PLCC	44PG/44PL
	E	40-DIP to 44-PLCC	40D/44PL-8051
	E	44-PGA to 44-PLCC	44PG/44PL
80C51XA	**E**	**40-DIP to 44-PLCC**	**44PG/44PL**
83C055 & 87C055	E	40-DIP to 42-Shrink DIP	40D/42SD6-83C055
8XC552, 8XC562 & 8XC592	E	68-PGA to 80-QFP, 2-piece surface mount	68PG/80QFR31-TOP-8XC552
	E	68-PLCC to 80-QFP, 2-piece surface mount	68PL/80QFR31-TOP-8XC552
83C751	E	24-DIP to 28-PLCC	24D3/28PL-751
83C752 & 87C752	E	28-PGA to 28-PLCC	28PG/PL
8XCE654	E	40-DIP to 44-QFP, 2-piece surface mount	40D/44QF31-TOP-8051 with 44QF31-SD
	E	44PGA to 44-QFP, 2-piece surface mount	44PG/44QF31-TOP with 44QF31-SD
	E	44-PLCC to 44-QFP, 2-piece surface mount	44PL/44QF31-TOP with 44QF31-SD
80CL31/51	E	40-DIP to 44-QFP, 2 piece surface mount	40D/44QF31-TOP-8051 with 44QF31-SD
	E	44-PGA to 44-QFP, 2 piece surface mount	44PG/44QF31-TOP with 44QF31-SD
	E	44-PLCC to 44-QFP, 2-piece surface mount	44PL/44QF31-TOP with 44QF31-SD
	E	40-DIP to 40-VSOP, 2 piece surface mount	40D/VS30-TOP with 40VS30-SD
P83CL167/168, P83CL267/268	E	64-DIP to 64-Shrink DIP, with solder tail pins	64D9/SD7-S
	E	64-Shrink DIP to 64-QFP, 2-piece surface mount	64SD7/QF39-TOP-83CL167 with 64QF39-SD
	E	68-PGA to 64-QFP, 2-piece surface mount	68PG/64QF39-TOP-83CL167 with 64-QF39-SD
	E	68-PLCC to 64-QFP, 2-piece surface mount	68PL/64QF39-TOP-83CL167 with 64QF39-SD
83CL580	E	68-PGA to 64-QFP, 2-piece surface mount	68PG/64QF39-TOP-83CL580 with 64QF39-SD
	E	68-PLCC to 64-QFP, 2-piece surface mount	68PL/64QF39-TOP-83CL580 with 64QF39-SD
	E	68-PGA to 56-VSOP, 2-piece surface mount	68PG/56VS__-TOP-83CL580
	E	68-PLCC to 56-VSOP, 2-piece surface mount	68PL/56VS__-TOP-83CL580
8XL51FA/FB	E	40-DIP to 44-LCC, 2-piece surface mount	40D/44LC-TOP-8051
	E	40-DIP to 44-QFP, 2-piece surface mount	40D/44QF31-TOP-8051 with 44QF31-SD
	E	44-PGA to 44-QFP, 2-piece surface mount	44PG/44QF31-TOP with 44QF31-SD
	E	44-PLCC to 44-QFP, 2-piece surface mount	44PL/44QF31-TOP with 44QF31-SD
85CL001	E	84-PGA to 84-PLCC	84PG/PL

Notes:

1. Applications: E = emulators, U = upgrade.
2. Programming adapters for 8051 architecture devices can be found in the programming accessory section of the EDI catalog. Adapters are available with monitoring posts for logic analyzers.

EMULATION TECHNOLOGY, INC.
Interconnect Solutions and PC Based Instrumentation

XA Microcontroller Development Tools

Development Tools

- ET-iCXA – Development Board
- ET-iCXA – In-Circuit Emulator
- ET-ASM-XA – Assembler
- ET-C-XA – C Compiler
- ET-CONV-XA – 8051 Code Converter
- ET-DEBUG-XA – Software Simulator
- ET-MP-51 – Programmer

ET-ASM-XA — Macro Assembler and Linker

The ET-ASM-XA assembler translates XA assembly language program into relocatable object code. The XA assembly language includes commands and directives specially designed to fit the XA architecture.

The two main tools of the package are assembler and linker programs. The linker enables the user to work with a number of modules and to locate the necessary segments. The ET-ASM-XA produces debug information that includes SYMB, LINE and FILE, and many more symbols.

The package also includes several utilities that convert from object code into Hex format. The assembler supports the XA special features like system and user modes, register banking and others.

The ET-ASM-XA assembler is integrated into a modern graphic interface that allows it to be used in a complete development environment.

ET-CONV-XA — 8051 Code Converter

The ET-CONV-XA converts assembly source and object code written for the 8051 to ET-ASM-XA. The ET-CONV-XA utilities make it possible to use code written in C, PLM or Assembler for the 8051 microcontroller and to adapt it to the XA architecture.

The CONV program first passes over the source file written for the 8051. It then translates the source file into ET-ASM-XA source file. In case of a conflict, the user will be asked by the program to select the right option. If the available code is different from the assembly source, the program converts the object file generated by high-level languages or assemblers into an ASCII file with all the possible assembly information.

ET-C-XA — C Compiler

The ET-C-XA is an ANSI C compiler with an extension designed to support XA special features. The compiler is compatible with other ANSI C compilers. The ET-C-XA is designed to make the code faster and smaller by using the special chip features. The ET-C-XA can support multi-tasking programs.

The system features ANSI C compatible, exception handling mechanism, interrupt handling mechanism, floating point variables, in-line assembler, extended C keyword for XA special architecture, special functions for checking memory integrity and more.

EMULATION TECHNOLOGY, INC.

Interconnect Solutions and PC Based Instrumentation

DB-XA — Development Board

DB-XA is a development board dedicated to all Philips XA microcontroller derivatives. It is serially linked to PC/XT/AT or compatible systems and can emulate the microcontroller using either the built-in clock oscillator or any other clock source connected to the microcontroller.

The system emulates the microcontroller in ROMless mode. A two microcontroller architecture leaves the serial port for user applications.

The software includes a Source Level Debugger for C and Assembler, On-line Assembler and Disassembler, Software Trace, Conditional Breakpoints and many other features. All the debugger functions run under DOS and Windows operating systems.

The code memory permits downloading and modifying of user's programs. Mapping the data memory to a target circuit or to the system is possible. Breakpoints allow real time execution until an opcode is executed at a specified address or line of the source code. All I/O lines are easily accessed and may be connected to the on-board switches and LEDs when trying out a specific idea. The system is supplied with a User's Manual, software, emulation cable and a power supply.

DS-XA — In-Circuit Emulator

This system is a real-time, fully transparent in-circuit emulator with 1 MByte mappable internal memory, 16M hardware breakpoints and real-time trace with trigger capabilities. The system emulates any XA derivatives in all the frequency ranges of the microcontroller.

ET-DEBUG-XA — Software Simulator

The ET-DEBUG-XA is a source level debugger for the XA architecture. The ET-DEBUG-XA enables fast and reliable program debugging at source level for ET-C-XA and ET-ASM-XA.

The ET-DEBUG-XA can execute your code on a target emulator or high-speed simulator. The software enables the user to follow and control code execution. The user can examine and change the data and the code.

ET-MP-51 — Programmer

All the necessary adapters and programming algorithms are added to the Et-MP-51 Programmer to support all the EPROM based XA derivatives.

2344 Walsh Avenue, Bldg. F, Santa Clara, CA 95051 USA Tel: (408)982-0660 Fax: (408)982-0664

HI-TECH SOFTWARE

HI-TECH C (XA) Product Brief

HI-TECH C Complier for the Philips XA microcontroller — technical specifications

HI-TECH Software's range of ANSI C embedded development systems now includes a product targeted specifically at Philips Semiconductors' XA (eXtended Architecture) microcontroller.

Key features of the XA development system include:

- Full ANSI C language supported
- Optimized code generation
- Full use of XA features and architecture
- Choice of memory models to suit differing applications
- Enhanced features including *bit* data type and *interrupt* functions
- Integrated development environment with project management facilities
- IEEE floating point with full maths library
- Macro assembler included
- Fully featured linker and librarian allow effective development of large projects
- Run-time library source code included.

System Requirements

HI-TECH C is available to run under MS-DOS, where it requires at least a 286 processor with 512k of conventional memory and 2MB of extended memory. It is also available for Unix systems running SunOs or Solaris on Sparc processors, or generic 386 Unix on 386 or higher processors.

ANSI/ISO C Language

HI-TECH C for the XA implements the ANSI/ISO standard for the C programming language. All data types are supported and have sizes and arithmetic properties as follows:

int	16 bits – signed and unsigned int available
char	8 bits – plain char is signed, *signed char* and *unsigned char* also available.
short	same an *int*
long	32 bits in signed and unsigned versions
float	32 bit IEEE floating point format – sign bit plus 8 bits of exponent plus 24 bits of mantissa.
double	same as *float*

All standard data structures are available, including *struct*, *union* and arrays. Structures may contain bitfields of up to 16 bits.

The compiler uses the XA stack for storage of automatic variables, so that functions are fully reentrant.

A full ANSI compatible library is provided, except for file handling functions. The library does include console I/O functions like *printf()* and *scanf()* which operate via simple serial port drivers included in the library, or through other user-supplied functions. Full library source code is provided to allow modification of this or other functions.

Enhanced Features

To maximize the effectiveness of C on the XA microcontroller, HI-TECH C offers some enhanced features, implemented in a manner which is in keeping with the style of the ANSI C standard.

Bit Variables

To allow access to the XA's bit handling functions the compiler implements *bit* variables. These are allocated in the bit addressable memory area (or may be mapped onto bit-addressable SFR's) and will be accessed with appropriate XA bit handling instructions.

Absolute Variables

SFR and other absolute variables may be defined with the absolute variable facility, e.g., the power control register PCON is defined as follows:

```
unsigned char PCON $0x404;
```

This defines the size, data type and address of the port. The compiler will not allocate space, but will use the specified address when accessing the variable. A header file is provided which defines the standard XA SFRs.

Interrupt Functions

Interrupt functions may be written completely in C by using the *interrupt* function feature. A function declared *interrupt* will contain code to save and restore any registers used, and will return with an appropriate **reti** instruction. Macros are provided to initialize and manage interrupt vectors, and enable and disable interrupts.

Near and Far Variables

To allow the programmer control over placement of variables, the keywords *near* and *far* are used to specify that a variable will be located in the on-board (directly addressable) RAM or in the indirectly addressable RAM respectively.

Non-Volatile RAM

Variables whose value is retained by battery-backed RAM when the microcontroller is turned off or reset can be defined with the *persistent* qualifier. Variables qualified persistent are allocated in a separate memory area, the address of which may be used-specified, and are not cleared to zero on startup like ordinary variables. Library functions are provided to initialize and validate the persistent memory area.

Memory Models

Because of the XA's Harvard architecture (separate RAM and ROM address spaces) and its ability to access memory in 64K segments, trade-offs are required between memory addressability and code size. To allow these trade-offs to be tuned to a particular application, three memory models are provided.

small	This model is designed for applications using less than 64K of code, and only directly addressable (on-board) RAM. Function calls use 16 bit addressing, and initialized data is stored in ROM rather than RAM.
medium	For applications using larger amounts of RAM, the medium model will give similar code efficiency to the small model at the expense of higher RAM usage. This model will use both directly and indirectly addressable RAM.
large	Where more than 64K code is required, the large model should be used. It uses 24 bit addressing to call functions, so that multiple 64K banks of code can be implemented in a transparent fashion.

In all memory models, the *far* and *near* keywords may be used to control RAM usage. In the large model, interrupt functions are automatically placed in bank 0 since interrupt vectors provide only a 16 bit address.

IEEE Floating Point

The compiler implements IEEE 32 bit floating point arithmetic. This has a range of $\pm 10^{\pm 38}$ and a precision of approximately 7 decimal digits. The floating point maths library includes the standard trigonometric, exponential, logarithmic and hyperbolic functions.

Advanced Optimization

HI-TECH C for the XA implements many compile-time optimization techniques to produce the smallest, fastest code possible. Some of the optimizations performed include:

- Constant folding – constant expressions (including floating point) are evaluated at compile time.
- Strength reduction – multiplication is reduced to shifting and adding where possible.
- Expression reordering – expressions are reordered and associative operators grouped to minimize complexity.
- Dynamic register allocation – registers are allocated to variables and temporaries based on function-wide analysis of variable usage.
- Common code elimination – where separate pieces of code produce common sequences, they are merged.
- Register parameters – function parameters are passed in registers where possible.

The set of optimizations performed is tuneable at compile time to allow trade-offs between compile time and code quality.

Macro Assembler Included

A full featured macro assembler is included with the compiler. This may be used to write separate assembler modules for use with C programs, or to write stand-alone assembler programs. Assembler code can also be embedded in-line in C modules. The compiler produces assembler code, and an assembled listing may be requested to allow inspection of the generated code.

Linker and Librarian

HI-TECH C includes a complete object code linker and librarian. This allows a large project to be split into multiple modules, thus making maintenance and recompilation easier. The linker combines a number of object modules, merging program code, data, etc., according to user-specified addresses of RAM and ROM.

The librarian allows object libraries to be built and maintained. The linker will extract from a library only those modules that satisfy currently unresolved external references. This allows a library to be used to hold multiple routines, not all of which may be used in any one project.

The linker produces a link map which provides information on code and data addresses and sizes for individual modules and the entire program.

Integrated Development Environment

The MS-DOS version of the HI-TECH C compiler includes an integrated development environment which provides a number of facilities to speed firmware development. These facilities include:

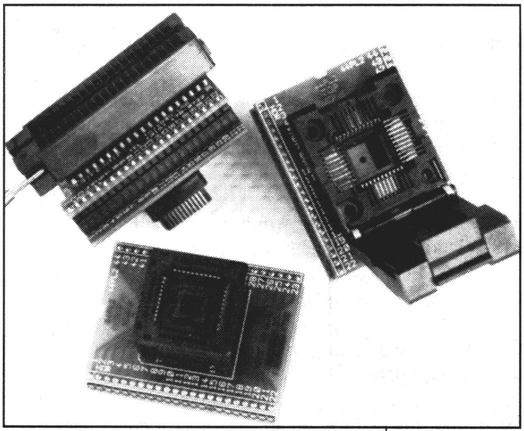
- Text editor, implementing Wordstar compatible and MS-Windows compatible key commands.
- C syntax colour coding.
- Compilation and linking control with automatic dependency analysis.
- Error reporting, with automatic location of errors, and automatic correction where possible, plus detailed error message explanation.
- Project management – a set of C and assembler source files, object files and libraries are defined with memory addresses, etc.
- String search across files.
- Multi-radix calculator.
- Summary memory usage map after project build.
- Library creation.
- Fully mouse and keyboard driven.
- User-defined commands for interfacing to external utilities such as EPROM programmers.

The integrated environment runs under MS-DOS, MS-Windows and Windows NT. There is also a command line driver provided. The Unix hosted compiler provides the command line driver only.

HI-TECH C resellers worldwide:

Phone	Fax	E-mail
Australia:	Hi-TECH Software	
(07) 300 5011	(07) 300 5346	hitech@hitech.com.au
USA:	Avocet Systems Inc.	
297 236 9055	207 236 6713	76570.1063@compuserve.com
	Macraigor Systems	
617 738 8197	617 739 8694	macraig@world.std.com
Japan:	Shoshin Corp.	
(03) 3270 5921	(03) 3245 0369	tokamoto@shoshin.co.jp
UK:	Pentica Systems	
(0734) 792 101	(0734) 774 081	
	System Science	
(071) 833 1022	(071) 837 6411	
Denmark:	Digitek Instruments	
+45 4342 4742	+45 4342 4743	
Italy:	Grifo	
051 892 052	051 893 661	
Switzerland:	Traco Electronic AG	
+41 1 284 2911	+41 1 201 1168	
Ireland:	Ashling Microsystems	
+353 61 33 4466	+353 61 33 4477	
France:	Convergie	
	+33 1 4789 0938	
Germany:	Reichmann Microcomputer	
7156 5635	7156 5141	100533.2535@compuserve.com
Sweden:	LinSoft AB	
13 11 1588	13 15 2429	

Adapters for Philips 51XA-G3



- **Transition 51XA-G3 devices into your 8051-FC products**
- **Program 51XA-G3 chips on your current programmer**

51XA-G3 adapters from Logical Systems are available for several purposes. Some allow programming of 51XA-G3 chips on your 87C51-FC programmer. Some are package converters that connect PLCC or QFP chips to the 51XA-G3 DIP footprint, primarily for device programming. Development and prototyping adapters include ones that connect 51XA-G3 DIP chips to PLCC footprints and ones that connect 51XA-G3 PLCC chips to '51-FC footprints. Availability is stock to 3 days. Pricing is from $65. - $149. 30-day satisfaction guaranteed.

51XA-G3 Package	Adapter Socket	Adapter Footprint	Adapter Purpose	Logical Systems AdapterPart Number
44 pin PLCC	Auto-eject	40 DIP, 51XA-G3	Programming	PA-XAG3-PD
44 pin PLCC	Lidded ZIF	40 DIP, 51XA-G3	Programming	PA-XAG3-PDZ
44 pin PLCC	Auto-eject	40 DIP, '51-FC	Programming	PA-XAG3FC-PD
44 pin PLCC	Lidded ZIF	40 DIP, '51-FC	Programming	PA-XAG3FC-PDZ
44 pin PLCC	Production	44 PLCC Plug '51-FC	Prototyping	PA-G3P-FCP
44 pin PLCC	Production	40 DIP, '51-FC	Prototyping	PA-G3P-FCD
44 pin QFP	Lidded ZIF	40 DIP, 51XA-G3	Programming & Prototyping	PA-XAG3-QD
44 pin QFP	Lidded ZIF	40 DIP '51FC	Programming	PA-XAG3FC-QD
44 pin QFP	Lidded ZIF	44 PLCC Plug 51XA-G3	Prototyping	PA44-QP
40 pin DIP	DIP ZIF	44 PLCC Plug 51XA-G3	Prototyping	PA-XAG3-DP
40 pin DIP	DIP ZIF	40 DIP, '51-FC	Prototyping	PA-XAG3FC-DD

DEVICE REFERENCE

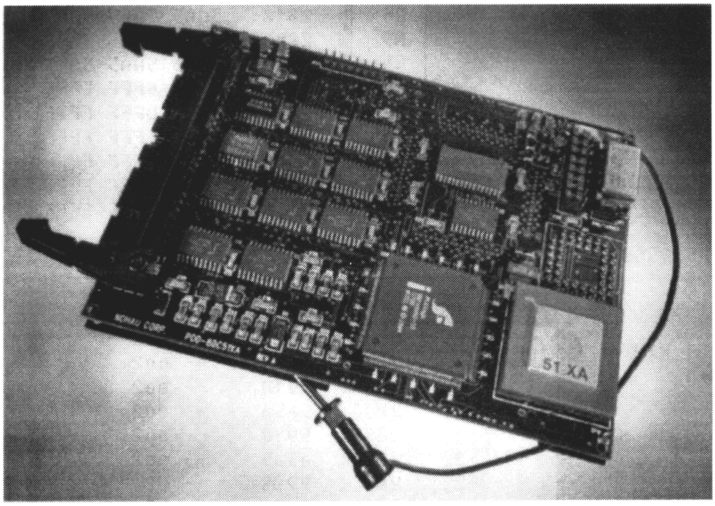
LOGICAL
SYSTEMS
P.O. Box 6184
Syracuse, NY 13217
Phone: 315.478.0722
FAX: 315.479.6753

Page 1 of 1

xaref.chp Rev 6/95

NOHAU
CORPORATION

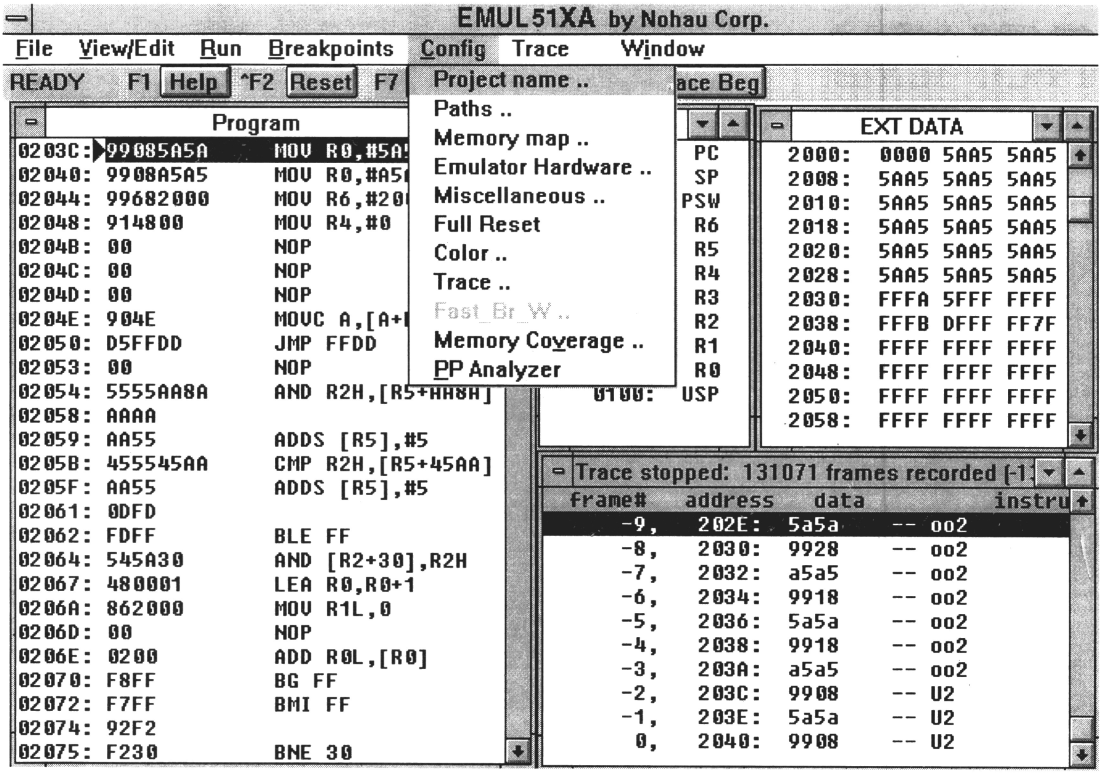
EMUL 51XA



In-Circuit Emulator for the Philips 80C51XA

Preliminary Product Information

- PC plug-in boards
- Uses Philips' bondout technology for accurate emulation
- Supports real time emulation
- Hardware breakpoint systems using RAM for High Flexibility
- Support for 256K combined code and data
- Hardware breakpoints on code
- Software breakpoints
- Supports real time trace, 104 bits wide (frame), 32K, or 128K frames deep
- Three trigger levels: Programmable Pre/Post trigger location, Ext address + data
- Time stamp
- Complex Trigger and Break
- Code coverage
- Tracing of the internal (not external) bus supported
- Microsoft Windows based user interface
- Availability of Register, Data, Program, Source, and Trace Windows
- Availability of on-line help
- Symbolic and Source Level Support
- Support for C variables
- Languages: Assembly and C
- 386 or better
- Workstation support through Nohau's LanICE (SUN, HP, etc.)


EMUL51XA by Nohau Corp.
File
View/Edit
Run
Breakpoints
Config
Trace
Window
READY
F1
Help
^F2
Reset
F7
ace Beg
Project name ..
Paths ..
Memory map ..
Emulator Hardware ..
Miscellaneous ..
Full Reset
Color ..
Trace ..
Fast_Br_W ..
Memory Coverage ..
PP Analyzer
Program
EXT DATA

```
0203C: 99085A5A     MOV R0,#5A
02040: 9908A5A5     MOV R0,#A5
02044: 99682000     MOV R6,#20
02048: 914800       MOV R4,#0
0204B: 00           NOP
0204C: 00           NOP
0204D: 00           NOP
0204E: 904E         MOVC A,[A+
02050: D5FFDD       JMP FFDD
02053: 00           NOP
02054: 5555AA8A     AND R2H,[R5+AA8A]
02058: AAAA
02059: AA55         ADDS [R5],#5
0205B: 455545AA     CMP R2H,[R5+45AA]
0205F: AA55         ADDS [R5],#5
02061: 0DFD
02062: FDFF         BLE FF
02064: 545A30       AND [R2+30],R2H
02067: 480001       LEA R0,R0+1
0206A: 862000       MOV R1L,0
0206D: 00           NOP
0206E: 0200         ADD R0L,[R0]
02070: F8FF         BG FF
02072: F7FF         BMI FF
02074: 92F2
02075: F230         BNE 30
```

```
        PC
        SP
        PSW
        R6
        R5
        R4
        R3
        R2
        R1
        R0
0100:   USP
```

```
2000:  0000 5AA5 5AA5
2008:  5AA5 5AA5 5AA5
2010:  5AA5 5AA5 5AA5
2018:  5AA5 5AA5 5AA5
2020:  5AA5 5AA5 5AA5
2028:  5AA5 5AA5 5AA5
2030:  FFFA 5FFF FFFF
2038:  FFFB DFFF FF7F
2040:  FFFF FFFF FFFF
2048:  FFFF FFFF FFFF
2050:  FFFF FFFF FFFF
2058:  FFFF FFFF FFFF
```

```
Trace stopped:  131071 frames recorded (-1
 frame#   address   data              instru
    -9,   202E:    5a5a      -- oo2
    -8,   2030:    9928      -- oo2
    -7,   2032:    a5a5      -- oo2
    -6,   2034:    9918      -- oo2
    -5,   2036:    5a5a      -- oo2
    -4,   2038:    9918      -- oo2
    -3,   203A:    a5a5      -- oo2
    -2,   203C:    9908      -- U2
    -1,   203E:    5a5a      -- U2
     0,   2040:    9908      -- U2
```

Product Availability:

The emulator is expected to be ready for beta testing in August, 1995.

Contact Information:

nohau
CORPORATION
51 E. Campbell Avenue, Campbell, CA 95008
PH: (408) 866-1820 FAX: (408) 378-7869

80C51XA
Software Development Tools

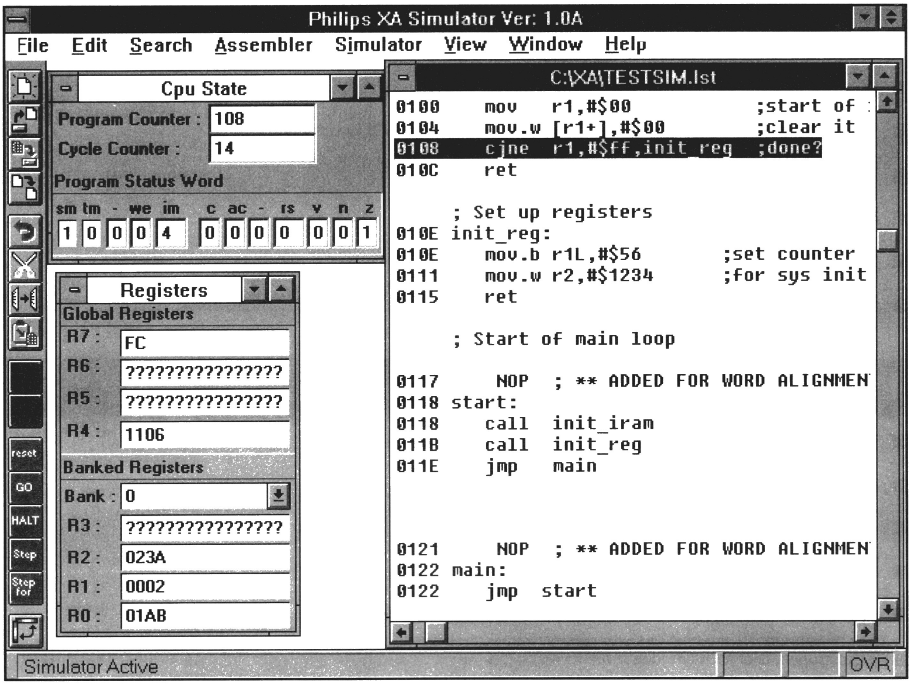
Software development tools for the XA

Philips provides standard software development tools for the XA family of microcontrollers. These software tools make up the core elements of an integrated development environment under Microsoft Windows. All three tools also work under DOS. They consist of a cross-assembler, an assembly language translator, and an architectural software simulator.

The cross-assembler provides a standard environment for XA programming. The translator allows 80C51 code to be ported to the XA. The software simulator acts as an immediately available test-bed for fresh code and provides several levels of errors and warnings.

All three of these tools operate on a PC under Windows or DOS. The programs include hooks for use with development tools from third-party vendors, including cross-assemblers, cross-compilers, and performance profilers.

The assembler/translator/simulator package is available free-of-charge from Philips, along with ensuing releases and updates, via a bulletin board system (800-451-6644). Each program incudes context-sensitive help and on-line documentation.


Philips XA Simulator Ver: 1.0A
File Edit Search Assembler Simulator View Window Help
Cpu State
Registers
C:\XA\TESTSIM.lst
reset
GO
HALT
Step
Step for
Simulator Active
OVR

Program Counter : 108
Cycle Counter : 14

Program Status Word

sm	tm	-	we	im	c	ac	-	rs	v	n	z
1	0	0	0	4	0	0	0	0	0	0	1

Global Registers
R7 : FC
R6 : ????????????????
R5 : ????????????????
R4 : 1106

Banked Registers
Bank : 0
R3 : ????????????????
R2 : 023A
R1 : 0002
R0 : 01AB

```
0100    mov   r1,#$00             ;start of :
0104    mov.w [r1+],#$00          ;clear it
0108    cjne  r1,#$ff,init_reg    ;done?
010C    ret

     ; Set up registers
010E init_reg:
010E    mov.b r1L,#$56         ;set counter
0111    mov.w r2,#$1234        ;for sys init
0115    ret

     ; Start of main loop

0117     NOP   ; ** ADDED FOR WORD ALIGNMEN
0118 start:
0118    call  init_iram
011B    call  init_reg
011E    jmp   main



0121     NOP   ; ** ADDED FOR WORD ALIGNMEN
0122 main:
0122    jmp  start
```

©1994 Macraigor Systems, Inc.

Philips
Semiconductors

PHILIPS

Standard XA Assembler

- Functions as an absolute assembler
- Supports macro development
- Available under Windows and DOS (real and protected modes)
- Defines factory-standard mnemonics
- Imposes no limit on size of source or object code files
- Supports a superset of IEEE 695 debug format
- Provides numerous directives for source code listing
- Outputs standard object files, list, and error files; compatible with various programmer's editors
- Provides well-documented output files
- Includes comprehensive, context-sensitive, on-line help

80C51 Assembly Code Translator

- Accepts standard 80C51 code (Intel and MetaLink)
- Supports all 80C51 derivatives
- Available under Windows and DOS (real and protected modes)
- Allows fast and efficient porting of 80C51 code libraries to the standard XA environment
- Non-invasive direct translation (one XA instruction for each 80C51 instruction)
- Flags NOPs, and problems associated with timing and code size along with possible optimization opportunities
- Provides multiple levels of warnings, errors, and hints
- Provides one-to-one listing of 80C51 input and resulting XA code
- Provides well-documented output files including 80C51 and XA source code listings; preserves labels and comments

XA Software Simulator

- Architectural simulator; covers CPU, one timer, and internal memory
- Available under Windows and DOS (protected mode)
- Supports numerous breakpoints, error highlighting, and cycle counting
- Resettable cycle counter allows simple timing analysis
- Provides hooks to support performance profiling of source code
- Reads superset of IEEE 695 debug file format
- Supports source-level debug for third party compilers

For more information call, 1-800-447-1500, ext. 1143

98-8080-820-01

©1994 Philips Electronics North America Corporation. Printed in U.S.A., SP8021/10M/SR1/1294

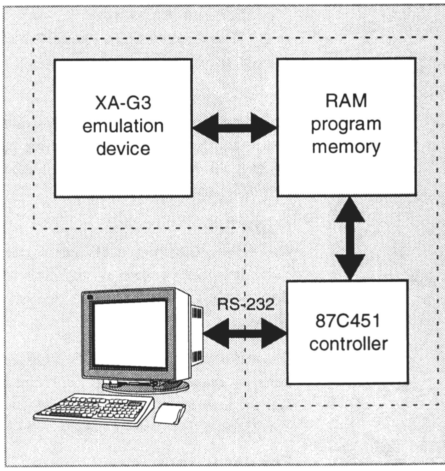
P51XA Development Board/Emulator

16 bit
XA
8 bit

The P51XA-DB/E supports the new 16-bit XA microcontroller family. It provides the user with an integrated hardware/ software development tool, allowing early verification of the prototype hardware and system software. The on-board EPROM programmer gives a quick and flexible approach to prototyping and debugging.

P51XA-DB/E Features:

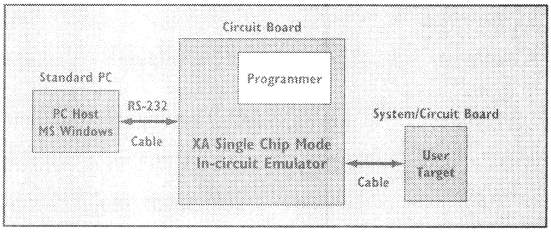
- Fully emulates the XA in single chip mode
  - Does not support external program memory modes
  - Allows external RAM for data storage
  - Bread board area for rapid development
- Integrated development environment for MS Windows
  - Source code editor
  - XA macro assembler
  - Translator (8051 to XA)
  - C compiler
  - Source code debug
- Allows single stepping or real-time execution
- 32K of emulation RAM for user program
- Allows display and/or modification of:
  - Registers, stack (user and supervisor)
  - PC, PSW
  - Data variables by name
  - Source, listing, disassembly
- Integrated EPROM XA programmer for PLCC package.



Philips Semiconductors

PHILIPS

P51XA Development Board/Emulator



The P51XA-DB/E is a development tool that supports the P51XAG3 microcontroller. The emulator will run a user program in real-time, and allow full source code debugging, including reading and writing all registers, RAM memory locations, and SFRs. The emulator is limited to debugging only single chip resources: the user may attach external RAM and ROM for use during real-time execution, but these areas cannot be debugged. The software included is a Microsoft Windows-based integrated development environment. It consists of an editor, translator (8051 to XA), assembler, simulator, C-Compiler, debugger and emulator interface. The translator is optimized to read an input file that is compatible with the MetaLink ASM51 assembler. Each line of the file is converted to an equivalent line of XA code. Pseudo-ops and directives are translated appropriately. The XA assembler is an absolute macro assembler. A debug file can be produced that is a superset of the IEEE 695 standard. The emulator and simulator interface is built into the development environment. Among the windows available are the user's source code and listing file, Watch Points, Breakpoints, Registers, CPU values, and SFRs. Any values in these windows may be modified by the user.

Specifications

Emulator Processors

The emulator has two microcontrollers. One is the emulation version of the XA-G3. The other is an 80C451 which communicates with the host computer and emulation device.

Host Characteristics

An IBM AT or compatible system running MS Windows 3.1 or later, one RS-232 port available for emulator communication.

Assembler

The included macro assembler uses factory-specified mnemonics. It is an absolute assembler and produces listing, debug (superset of IEEE 695), hex and error files. Two versions are supplied, a DOS command line version and the Windows environment version.

Compiler

The C-Compiler supports up to 64K bytes of code and data (small memory model). The compiler produces executable code rather than relocatable object modules that must be linked.

Translator

The included translator will read ASM51 or similar assembly language source and output XA source code. The translation is done on a line-by-line basis and is an aid in porting code between the 8051 family and the XA. It is supplied in DOS command line and Windows environment versions.

Emulator Interface

The emulator interface allows full source code and symbolic debugging of assembler or compatible high-level languages. The interface reads in either hex object code or a superset of IEEE 695 debug file for symbolic and source information.

All variables may be displayed by name and type. The user may choose between hex, ASCII, and decimal displays in most windows.

Breakpoints

The user may specify up to ten breakpoints. These are used during real-time emulation. Breakpoints are not inserted during single stepping.

Frequency

The emulator uses an on-board TTL-type canned oscillator to run the XA emulation chip. The oscillator supplied with the emulator operates the XA at 20 MHz. The user may replace this with another canned oscillator to run at any desired speed up to 20 MHz.

Built-in Programmer

The built-in programmer may be used to program an EPROM version of the XA-G3. All programming features are supported.

Emulation Restrictions

- The user's code may not use the Trace or Breakpoint interrupts or vectors
- Eight words of system stack are used by the emulator
- Debug on-chip code and data

Warranty

Six months limited warranty, parts and labor.

Items Supplied

The emulator comes as a complete package including the development board/emulator, emulation cable with 44-pin PLCC plug, power supply, 9-pin RS-232 interface cable with 25-pin host adapter, 120/220 volt power supply, built-in programmer, quick-start guide and all manuals on line.

Ordering Information

Part No:	P51XA-DBE SD
Order No:	9352-041-50112

For more information call your local Philips Office or:

North America: Call 1-408-991-51XA (5192)
Europe Fax: 31-40-722277
Asia Fax: 886-2-382-4484

Now supports XA!

Universal In-Circuit Emulator for 8051/31 Series

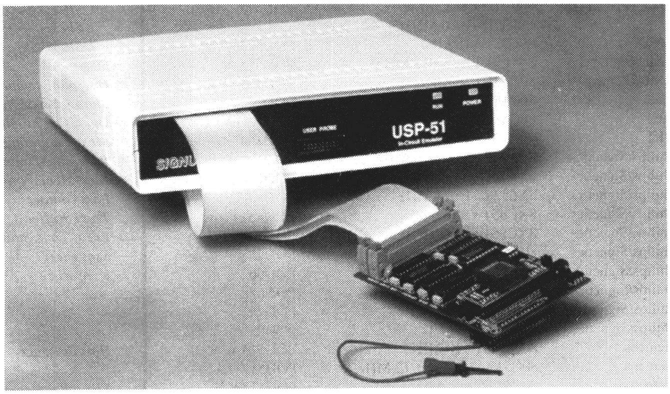
USP-51 Key Features

- Memory display or edit while your code is executing in real-time
- View the trace during execution
- HLL Debug for C-51 and PL/M-51
- Pass-point to monitor internal RAM, variables, and registers while running
- Real-time transparent emulation up to 40 MHz
- 32K Frame by 80-bit execution trace buffer with time stamp
- In-line symbolic assembler and disassembler
- Up to 256K of emulation overlay program RAM with bank switching
- Memory map in 256 byte blocks
- Up to 256K of real-time hardware breakpoints
- Three complex events for trace, sample trace, or break
- Two 16-bit pass counters
- 8 Level hardware event sequencer
- 8 Channel user logic state analyzer
- External trigger input and outputs
- Performance analysis histogram
- Wide range of uP pods to emulate most 8051 family members
- Windowed and command line user interface
- 115 K-baud serial download, (64K program downloads in 14 sec.)

User Interface

Designed to work with DOS, IBM-AT, 386/486 compatible computers, the window/menu user interface gives you:

- Pop-up windows for source, registers program, SFR's, trace, stack, setup, symbols, locals, and variables watch
- HLL windows for C-51 and PL/M-51 for source level debugging
- You may define your own watch window for complex variables like arrays and structures
- 132 by 60 Display, EGA, VGA
- Full screen edit with mouse support
- User defined SFR window
- Extensive macro program support
- You can define, save, and recall trigger setups, breakpoints, trace, and complex events to or from disk
- Locals window displays all local variables automatically



Complex Events

A complex event is a set of conditions that may be used to qualify emulation breakpoints, event sequencer, or trace filtering in real time. The system has three complex event triggers available that may be used for the following:

- 256K Address breakpoints including within and outside address ranges
- 16-Bit data pattern with less than, greater than, equal, not equal, and don't care combinations.
- Qualify on RD, WR, Int, instruction fetch, and operand read
- External input with programmable trigger polarity

All events may be count qualified or delayed by two 16-bit pass counters. Events also work with the eight level sequencer to trigger from any set of events or pass count condition.

Breakpoints

Breakpoints are used to stop user program execution while preserving the current program status. They may be combinations of:

- Register values and internal RAM
- Addresses and address ranges
- Complex events
- Pass counts
- Sequenced events
- Trace buffer full
- External trigger input

Trace Buffer

The system trace buffer is a high speed RAM that captures activity of the microprocessor internal bus and pins in real time. A trace start/stop switch allows you to filter unwanted information from the captured data. The trace will store up to 32K samples, (80 bit frames), comprised of the following:

- Address Bus
- Data Bus
- Control signals
- I/O Pins
- Time Stamp
- User logic state input, (8 bits)

The trace may be started and stopped by any combination of:

- Complex events
- Pass counts
- Event sequences
- Go command
- Trace full condition

The trace buffer also has a frame counter to stop tracing after a specified number of frames have been captured. This means you may capture as much as 32K of small program fragments at full target speed.

The trace contents can be viewed during program execution without stopping or slowing down the prototype microcontroller.

Made and supported in the U.S.A.

SIGNUM SYSTEMS CORP., *171 E. Thousand Oaks Blvd., Thousand Oaks, CA 91360*

Corp tel: (805) 371-4608 • FAX (415) 371-4610 • EMail Attmail!Signum • Sales tel: (415) 903-2220 • FAX (415) 903-2221

Now supports XA!

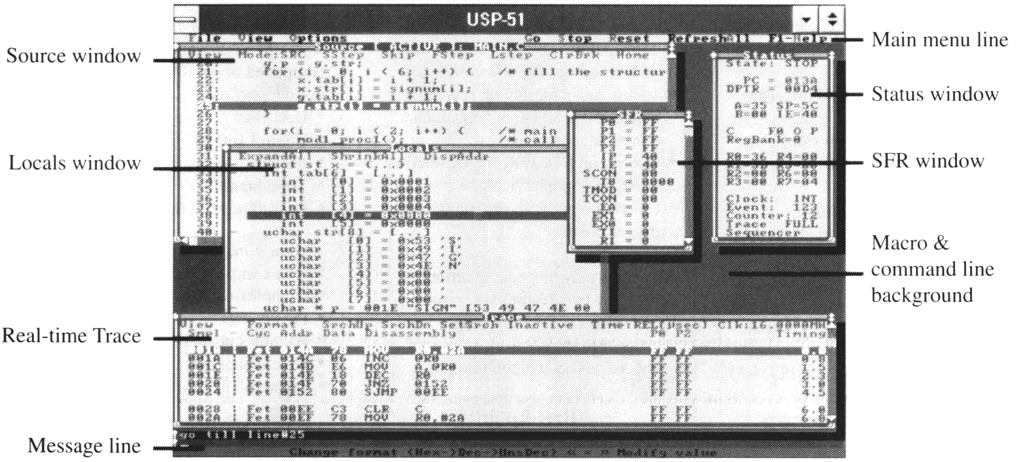
SPECIFICATIONS

Model USP-51

Mfrs.	Target Processors	In-Circuit Pod
All	8031 80C32 80C51FA 80C154 • 20 /30/40 MHz	POD31
All	80C31/32 8XC51/52 85C154 • 16 MHz	POD51
AMD	80C321 • 16 MHz	POD321
AMD-Siemens	80515 - 80535 • 12 MHz	POD515
AMD Siemens	80C535 • 16 MHz	POD535
Intel	80C152JA/JB/JC/JD • 16 MHz	POD152
Intel	80C51GB • 16 MHz	POD51GB
OKI	85C154 • 30 MHz	POD154
Philips/Signetics	8XC552 • 16/20/24 MHz	PODX552
Philips/Signetics	8XC562 • 16/20/24 MHz	PODX562
Philips/Signetics	8X652/654 • 24 MHz	PODX652
Philips/Signetics	8XC451 • 16 MHz	POD451
Philips/Signetics	8XC851 • 16 MHz	PODX851
Philips/Signetics	80C552 • 16/24 MHz	POD552
Philips/Signetics	80C562 • 16/24 MHz	POD562
Philips/Signetics	80C652/654 • 16/24 MHz	POD652
Philips/Signetics	8XC751 • 16 MHz	POD751
Philips/Signetics	80C851 • 16 MHz	POD85
Siemens	80C515A • 18 MHz	POD515A
Siemens	80C517 - 80C537 • 12 MHz	POD517
Siemens	80C517A • 18 MHz	POD517A
Siemens	C501 • 40 MHz	POD501
Siemens	80C537 • 12 MHz	POD537
Silicon Systems	K246 • 16 MHz	POD246

In-Circuit Emulators

Maximum emulation speed	
USP-51	*20 MHz*
USP-51-30	*30 MHz*
USP-51-40	*40 MHz*
USP-51-SS	*30 MHz (Silicon Systems)*
Size	*260 x 260 x 64 mm*
Operating temperature	*0 to 40 C*
Storage temperature	*-10 to 65 C*
Operating humidity	*0 to 90%*
Max. Emulation Program Memory	*256K*
Program Memory Mapping	*256 byte boundary*
Data Memory Mapping	*256 byte boundary*
Pass counters	*2 each, 16 bit*
Trace Buffer	*32 Kframes by 80 bits*
Event Time Stamp	*32 bits, 100 ns resolution*
Sequencer	*8 level hardware*
User probe	*8 channel logic input* *1 trigger input with gate* *6 trigger outputs (Events, Pass Counters, Sequencer)*
Host interface	*Asynchronous RS-232C, 9600-115Kbaud, XON/XOFF*
File upload/download format	*Intel HEX/AOMF, 2400AD, Archimedes, BSO/Tasking, Franklin, Keil*



USP-51 Screen Example

SIGNUM SYSTEMS

171 E. Thousand Oaks Blvd
Thousand Oaks, CA 91360
Tel: (805) 371-4608
FAX: (805) 371-4610

EMail: Attmail!Signum

800 El Camino Real
Mountain View, CA 94040
Tel: (415) 903-2220
FAX: (415) 903-2221

Now supports XA!

SYSTEM GENERAL

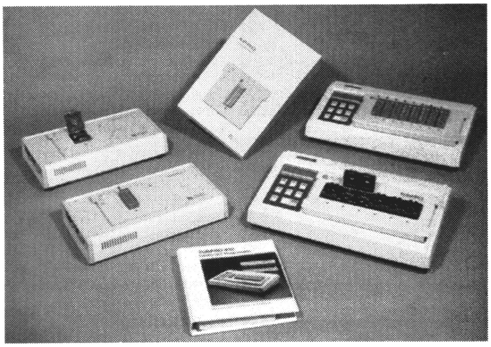
Universal Device Programmers

SG Family of Universal Device Programmers

- * Full range of programming support for Low Voltage devices from all manufacturers
- * Engineering and Production Programmers for EPROMs, EEPROMs, FLASH EPROMs, PROMs, Microcontrollers, PLDs, EPLDs, PALs, GALs, PEELs, FPLAs, FPGAs, etc.
- * Stand-alone & PC-remote Programmers
- * AC/DC Parametric testing to provide leakage test for ESD
- * Unique patented Features Auto-sense and Turbo-mapping provide extremely high throughput and optimize yield
- * Patented pin-driver technology
- * Fully certified by most major Semiconductor manufacturers
- * Programmer of choice by many of the IC Distributors at their "Value-Added Programming Centers"
- * Interface to automated handling equipment
- * Responsive, reliable technical support
- * Lifetime Free SW Updates Available Via BBS

Contact:
SYSTEM GENERAL CORPORATION
1603 A South Main Street
Milpitas, CA 95035
Phone: 1-800-967-4 PRO (1-800-967-4776)
(408) 263-6667
Fax: (408) 262-9220

SYSTEM GENERAL



System General has been manufacturing Device Programmers since 1985. In 1989 we began delivering IC Manufacturer approved solutions to the U.S. Market. Having become the Number 1 Programmer Manufacturer in Asia, we are now considered one of the "Big 3" programmer manufacturers in the world by the Semi-Houses. We have been meeting high volume requirements in harsh environments since inception. Every programmer we make must pass a rigid burn-in procedure before being shipped out.

We offer a full range of programming solutions, so one is bound to be the ideal fit for your application. Whether you are a design engineer working on the most basic EPROM or PLD development, or a manufacturing engineer with a high volume requirement for programming Memory or Logic devices, we have a very cost effective solution for you. We have also been known to custom design our software to customer requirements.

SG's excellent device support and technical support are widely renowned in the semiconductor industry. In most cases we will be the first programmer manufacturer to support a new device release. IC Manufacturer certification means that all the Semi-Houses have SG test units to help support our customers. Our customer references are the best in the business!

Every System General programmer comes standard with our easy-to-use DOS-based control SW, all the cabling you need to get started, and an anti-static ground strap.

Call System General to find out more about the programmers that are fast becoming THE NEW industry standard!

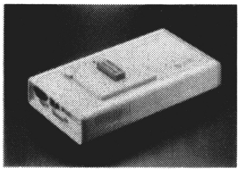
Ordering information (See reverse)

APRO	Single socket Memory Programmer	$ 895.00
APRO⁺	As above, includes CMOS EPLDs	$1295.00
Turpro-1	Universal IC-Programmer (Level 1)	$1695.00
Turpro-1/FX	Universal Production Programmer (Level 1)	$2495.00
Turpro-832	Gang EPROM Programmer	$1995.00
Turpro-832	(Set) Gang/Set EPROM Programmer	$2695.00
Turpro-840	Gang/Set EPROM/Micro Programmer	$3995.00

Now supports XA!

SYSTEM GENERAL

P.O. Box 361898, Milpitas, CA 95036-1898 U.S.A.
Tel: 408-263-6667/Fax: 408-262-9220/1-800-967-4776



TURPRO-1/TX

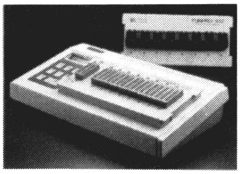
- Universal programmer plus DC parametric tester for Memory and Logic devices
- Unique leakage current test to ensure device quality
- Interface to Unix, Apple, NEC under VT100 modem mode
- High Speed pin driver for future High Speed PLD.

TURPRO-1/FX

- Universal Production Pin-Driven IC Programmer
- Drop-in Replacement for Data I/O Handler Interface
- Supports all Device Technologies and Packages to 84 Pins and beyond
- Vector Tests PLDs to 44 Pins
- Parallel Port File Downloads for High Speed Transfer

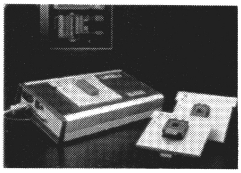
TURPRO-840

- Universal Memory/Micro Gang/Set Programmer, 8 at a Time
- Supports 24, 28, 32, 40 & 44 Pin Devices in DIP/PLCC
- Stand-Alone and/or Computer Remote Via RS232C From PC
- Parallel Port File Downloads for High Speed Transfer
- Full Editor in Remote
- Device Auto-ID, Easy-To-Use



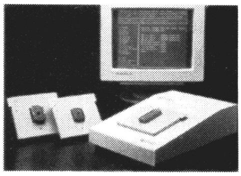
TURPRO-832

- Universal Memory Gang/Set Programmer, 8 at a Time
- Supports 24, 28 & 32 Pin Devices in DIP/PLCC
- Stand-Alone and/or Computer Remote Via RS232C From PC
- Parallel Port File Downloads for High Speed Transfer
- Full Editor in Remote
- Device Auto-ID, Easy-To-Use



TURPRO-1

- Universal Engineering Pin-Driven IC Programmer
- Drop-In Replacement for Data I/O
- Supports All Device Technologies and Packages to 84 Pins and beyond
- Vector Test PLDs to 44 Pins



APRO

- Universal Engineering Memory/Micro Programmer
- Supports 24, 28, 32, 40 & 44 Pin Devices In DIP/PLCC
- Computer Remote Via RS232C From PC at 115.2K Baud
- Full Editor
- Device Auto-ID, Easy-To-Use
- Only $895. Optional EPLD Support for $1,295

System General Product Comparison Chart

	APRO	T-832		T-840	T-1	T-1/FX
		GANG	SET			
Programmer User Interface:						
Standalone Programming Operations	No	Yes	Yes	Yes	No	No
Must have PC to Operate	Yes	No	No	No	Yes	Yes
RS232 for Computer Remote Control	Yes	Opt	Yes	Yes	Yes	Yes
PC Interface Software	Yes	Opt	Yes	Yes	Yes	Yes
Parallel Port for High Speed Transfer	No	Opt	Yes	Yes	No	Yes
Operates with Device Handler	No	No	No	Yes	No	Yes
Programmer Device Support:						
Standard on-board RAM (Bits)	8 MEG	N/A	8 MEG	8 MEG	256K	8 MEG
E/EPROMs (24,28 & 32-pin DIP)	Yes	Yes	Yes	Yes	Yes	Yes
FLASH EPROMS to 32-pin DIP	Yes	Yes	Yes	Yes	LEVII	LEVII
EPROMs 40-pin, 16bit-wide	Yes	No	No	Yes	Yes	Yes
GANG E/EPROMs to 32-pin DIP	No	Yes	Yes	Yes	No	No
"SET" E/EPROMs to 32-pin DIP	No	Opt	Yes	Yes	No	No
GANG/SET Meomories to 40-pin DIP	No	No	No	Yes	No	No
Sequential "SET" Programming	Yes	N/A	N/A	N/A	Yes	Yes
40-pin (Intel) Micros	Yes	No	No	Yes	Yes	Yes
Motorola Micros	No	No	No	No	No	Yes
Bipolar RPOMs to 24-pin DIP	No	No	No	No	LEVII	LEVII
PLDs to 28-pin DIP	No	No	No	No	LEV I	LEV I
PLDs to 40-pin DIP	No	No	No	No	LEVII	LEVII
PLDs to 84-pin (& beyond)	No	No	No	No	LEVIII	LEVIII
Single PLCC Device Options	Yes	Opt	Yes	Yes	Yes	Yes
GANG PLCC E/EPROM Options	N/A	Opt	Yes	Yes	N/A	N/A
Programmer Architecture:						
Pin Driver Architecture	Yes	Yes	Yes	Yes	Yes	Yes
Turbo Mapping Technology	Yes	Yes	Yes	Yes	Yes	Yes
Fast EPROM Programming Speeds	Yes	Yes	Yes	Yes	No	Yes
Auto Sense Feature	Yes	Yes	Yes	Yes	Yes	Yes
Programmer Price:	$895	$1995	$2695	$3995	$1695	$2495
Level Upgrade	N/A	$ 700	N/A	N/A	$600 EA	$800 EA

Call Us Now For Turnkey Solutions

Section 7

Package Information

CONTENTS

Soldering 505

SDIP42: plastic shrink dual in-line package; 42 leads (600 mil) SOT270-1 507

LQFP44: plastic low profile quad flat package; 44 leads; body 10 x 10 x 1.4 mm SOT389-1 508

PLCC44: plastic leaded chip carrier; 44 leads SOT187-2 509

44-pin CerQuad J-Bend (K) Package 1472A 510

INTRODUCTION

There is no soldering method that is ideal for all IC packages. Wave soldering is often preferred when though-hole and surface mounted components are mixed on one printed-circuit board. However, wave soldering is not always suitable for surface mounted ICs, or for printed-circuits with high population densities. In these cases reflow soldering is often used.

This text gives a very brief insight to a complex technology. A more in-depth account of soldering ICs can be found in our *"IC Package Databook"* (order code 9398 652 90011).

THROUGH-HOLE MOUNTED PACKAGES

Table 1. Types of through-hole mounted packages

TYPE	DESCRIPTION
DIP	plastic dual in-line package
SDIP	plastic shrink dual in-line package
HDIP	plastic heat-dissipating dual in-line package
DBS	plastic dual in-line bent from a single in-line package
SIL	plastic single in-line package

Soldering by dipping or wave

The maximum permissible temperature of the solder is 260°C; solder at this temperature must not be in contact with the joint for more than 5 seconds. The total contact time of successive solder waves must not exceed 5 seconds.

The device may be mounted to the seating plane, but the temperature of the plastic body must not exceed the specified maximum storage temperature ($T_{stg\ max}$). If the printed-circuit board has been pre-heated, forced cooling may be necessary immediately after soldering to keep the temperature within the permissible limit.

Repairing soldered joints

Apply a low voltage soldering iron (less than 24V) to the lead(s) of the package, below the seating plane or not more than 2mm above it. If the temperature of the soldering iron bit is less than 300°C it may remain in contact for up to 10 seconds. If the bit temperature is between 300 and 400°C, contact may be up to 5 seconds.

SURFACE MOUNTED PACKAGES

Table 2. Types of surface mounted packages

TYPE	DESCRIPTION
SO	plastic small outline package
SSOP	plastic shrink small outline package
TSSOP	plastic thin shrink small outline package
VSO	plastic very small outline package
QFP	plastic quad flat package
LQFP	plastic low profile quad flat package
SQFP	plastic shrink quad flat package
TQFP	plastic thin quad flat package
PLCC	plastic leaded chip carrier

Reflow soldering

Reflow soldering techniques are suitable for all SMD packages, ease of soldering varies with the type of package as indicated in Table 3.

The choice of heating method may be indluenced by larger plastic packages (QFP or PLCC with 44 leads, or more). If infrared or vapor phase heating is used and the large packages are not absolutely dry (less than 0.1% moisture content by weight), vaporization of the small amount of moisture in them can cause cracking of the plastic body. For more information on moisture prevention, refer to the Drypack chapter in our *"Quality Reference Manual"* (order code 9398 510 63011).

Reflow soldering requires solder paste (a suspension of fine solder particles, flux and binding agent) to be applied to the printed-circuit board by screen printing, stenciling or pressure-syringe dispensing before package placement.

Several techniques exist for reflowing; for example, thermal conduction by heated belt. Dwell times vary between 50 and 300 seconds depending on heating method. Typical reflow temperatures range from 215 to 250°C.

Preheating is necessary to dry the paste and evaporate the binding agent. Preheating duration: 45 minutes at 45°C.

Table 3. Suitability of surface mounted packages for various soldering methods
Rating from ‘a’ to ‘d’: **‘a’ indicates most suitable** (soldering is not difficult); **‘d’ indicates least suitalbe** (soldering is achievable with difficulty).

TYPE	REFLOW METHOD					DOUBLE WAVE METHOD
	INFRARED	HOT BELT	HOT GAS	VAPOR PHASE	RESISTANCE	
SO	a	a	a	a	d	a
SSOP	a	a	a	c	d	c
TSSOP	b	b	b	c	d	d
VSO	b	b	a	b	a	b
QFP	b	b	a	c	a	c
LQFP	b	b	a	c	d	d
SQFP	b	b	a	c	d	d
TQFP	b	b	a	c	d	d
PLCC	c	b	b	d	d	b

Wave soldering

Wave soldering is **not** recommended for SSOP, TSSOP, QFP, LQFP, SQFP or TQFP packages. This is because of the likelihood of solder bridging due to closely-spaced leads and the possibility of incomplete solder penetration in multi-lead devices.

If wave soldering cannot be avoided, the following conditions must be observed:

- A double-wave (a turbulent wave with high upward pressure followed by a smooth laminar wave) soldering technique should be used.
- For SSOP, TSSOP and VSO packages, the longitudinal axis of the package footprint must be parallel to the solder flow **and** must incorporate solder theives at the downstream end.
- For QFP, LQFP and TQFP packages, the footprint must be at and angle of 45° to the board direction **and** must incorporate solder thieves downstream and at the side corners.

Even with these conditions, only consider wave soldering for the following package types:

- SO
- VSO
- PLCC
- SSOP **only with body width 4.4mm**, e.g., SSOP16 (SOT369-1) or SSOP20 (SOT266-1).
- QFP **except** QFP52 (SOT379-1), QFP100 (SOT317-1, SOT317-2 and SOT382-1) and QFP160 (SOT322-1); these are **not** suitable for wave soldering.
- LQFP **except** LQFP32 (SOT401-1), LQFP48 (SOT313-1, SOT313-2), LQFP64 (SOT314-2), LQFP80 (SOT315-1); these are **not** suitable for wave soldering.
- TQFP **except** TQFP64 (SOT357-1), TQFP80 (SOT375-1) and TQFP100 (SOT386-1); these are **not** suitable for wave soldering.

SQFP are **not** suitable for wave soldering.

During placement and before soldering, the package must be fixed with a droplet of adhesive. The adhesive can be applied by screen printing, pin transfer or syringe dispensing. The package can be soldered after the adhesive is cured.

Maximum permissible solder temperature is 260°C, and maximum duration of package immersion in solder is 10 seconds, if cooled to less than 150°C within 6 seconds. Typical dwell time is 4 seconds at 250°C.

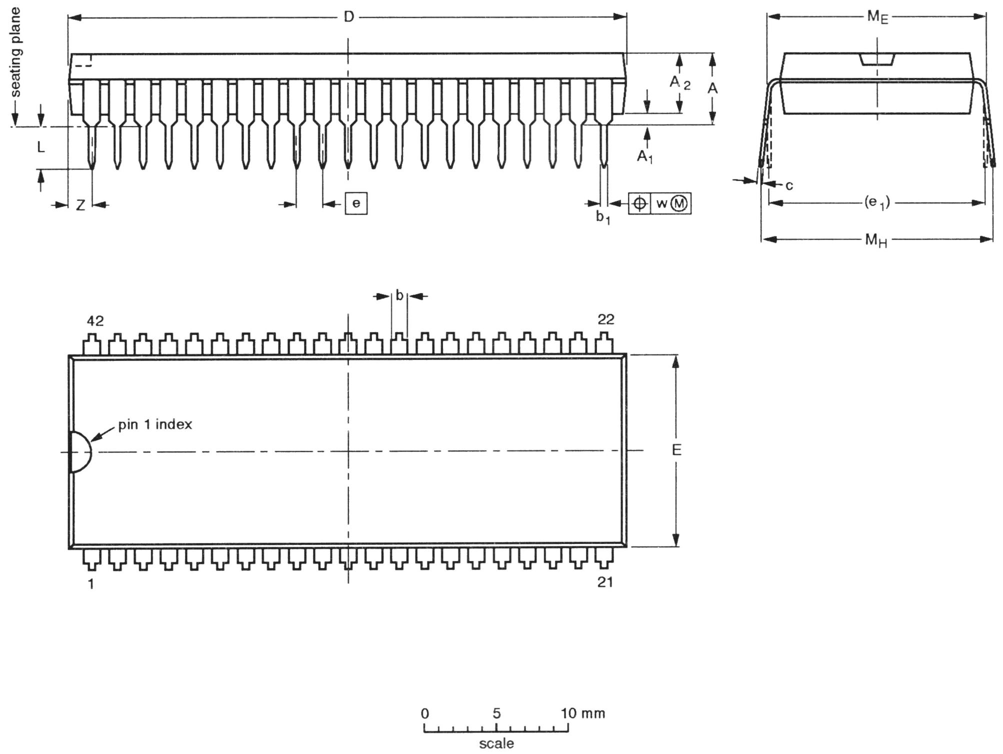
A mildly-activated flux will eliminate the need for removal of corrosive residues in most applications.

Repairing soldered joints

Fix the component by first soldering two diagonally-opposite end leads. Use only a low voltage soldering iron (less than 24V) applied to the flat part of the lead. Contact time must be limited to 10 seconds at up to 300°C. When using a dedicated tool, all other leads can be soldered in one operation within 2 to 5 seconds at between 270 and 320°C.

Package outlines

SDIP42: plastic shrink dual in-line package; 42 leads (600 mil) SOT270-1


seating plane
D
A2
A
A1
L
Z
e
b1
w M
ME
c
(e1)
MH
b
42
22
pin 1 index
E
1
21
0
5
10 mm
scale

DIMENSIONS (mm are the original dimensions)

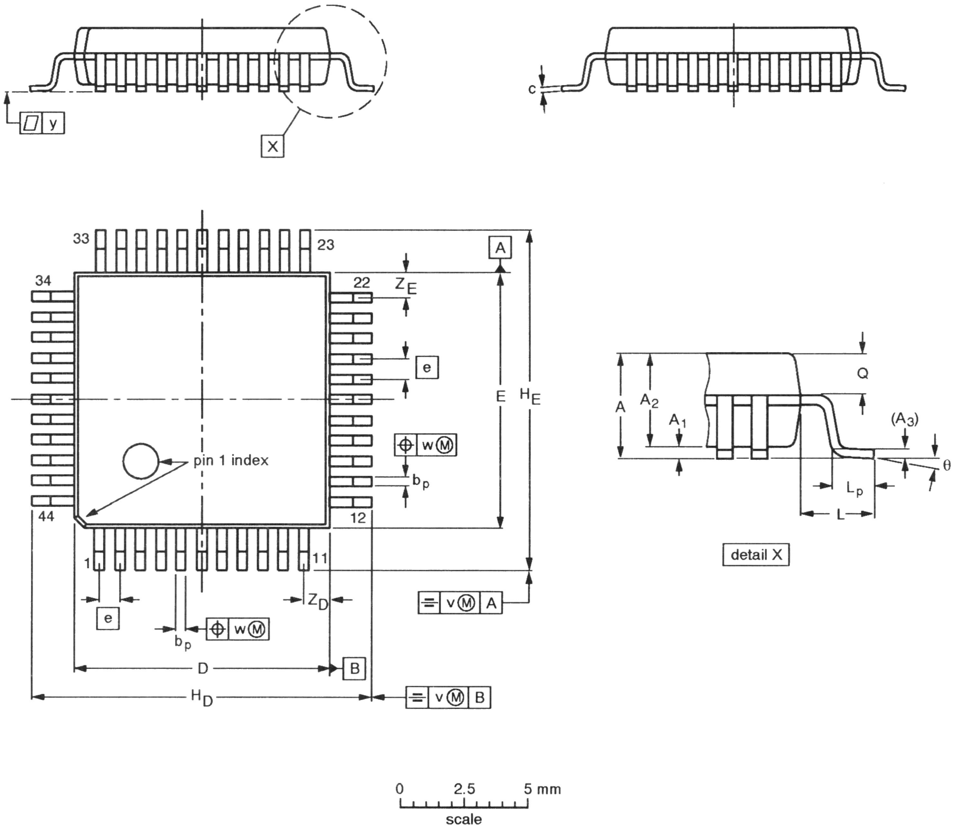
UNIT	A max.	A_1 min.	A_2 max.	b	b_1	c	$D^{(1)}$	$E^{(1)}$	e	e_1	L	M_E	M_H	w	$Z^{(1)}$ max.
mm	5.08	0.51	4.0	1.3 0.8	0.53 0.40	0.32 0.23	38.9 38.4	14.0 13.7	1.778	15.24	3.2 2.9	15.80 15.24	17.15 15.90	0.18	1.73

Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT270-1						~~90-02-13~~ 95-02-04

LQFP44: plastic low profile quad flat package; 44 leads; body 10 x 10 x 1.4 mm SOT389-1


y
X
c
33
23
34
22
Z_E
A
e
E
H_E
w M
pin 1 index
b_p
44
12
1
11
Z_D
e
b_p
w M
v M A
D
B
H_D
v M B
Q
A
A_2
A_1
(A_3)
θ
L_p
L
detail X
0
2.5
5 mm
scale

DIMENSIONS (mm are the original dimensions)

UNIT	A max.	A_1	A_2	A_3	b_p	c	D(1)	E(1)	e	H_D	H_E	L	L_p	Q	v	w	y	Z_D(1)	Z_E(1)	θ
mm	1.60	0.15 0.05	1.45 1.35	0.25	0.45 0.30	0.20 0.12	10.10 9.90	10.10 9.90	0.80	12.15 11.85	12.15 11.85	1.0	0.75 0.45	0.70 0.57	0.20	0.20	0.10	1.14 0.85	1.14 0.85	7° 0°

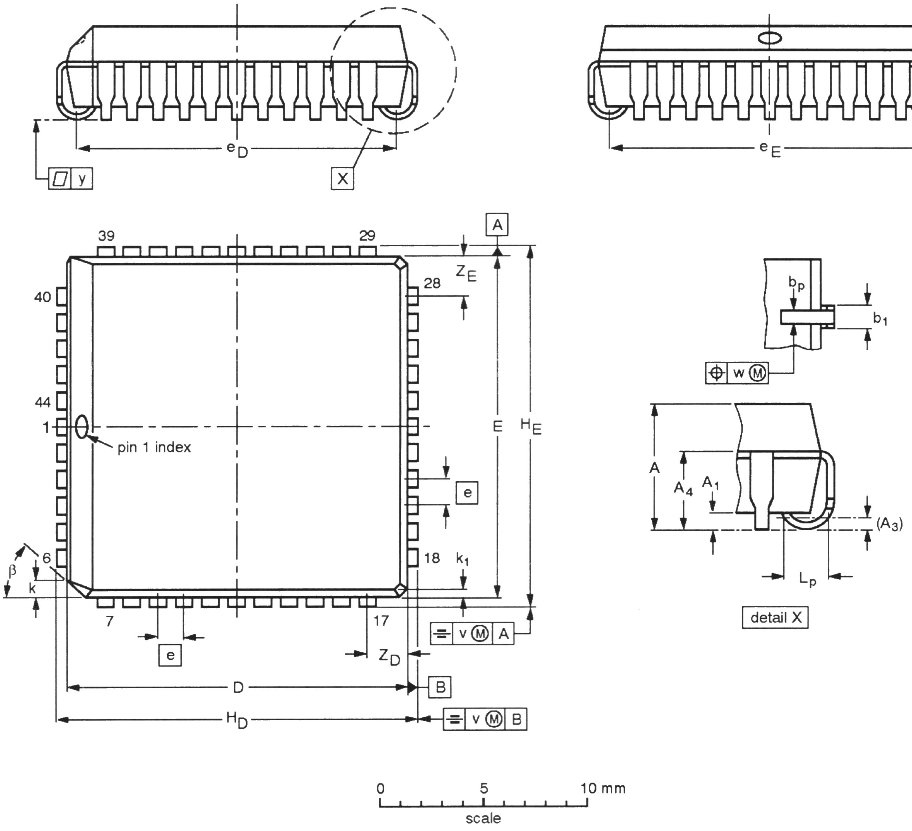
Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT389-1						95-02-25

Package outlines

PLCC44: plastic leaded chip carrier; 44 leads SOT187-2



e_D
y
X
e_E
A
39
29
Z_E
28
40
44
1
pin 1 index
E
H_E
e
18
k_1
6
β
k
7
17
v Ⓜ A
e
Z_D
D
B
H_D
v Ⓜ B
b_p
b_1
w Ⓜ
A
A_4
A_1
(A_3)
L_p
detail X
0
5
10 mm
scale

DIMENSIONS (millimetre dimensions are derived from the original inch dimensions)

UNIT	A	A_1 min.	A_3	A_4 max.	b_p	b_1	$D^{(1)}$	$E^{(1)}$	e	e_D	e_E	H_D	H_E	k	k_1 max.	L_p	v	w	y	$Z_D^{(1)}$ max.	$Z_E^{(1)}$ max.	β
mm	4.57 4.19	0.51	0.25	3.05	0.53 0.33	0.81 0.66	16.66 16.51	16.66 16.51	1.27	16.00 14.99	16.00 14.99	17.65 17.40	17.65 17.40	1.22 1.07	0.51	1.44 1.02	0.18	0.18	0.10	2.16	2.16	45°
inches	0.180 0.165	0.020	0.01	0.12	0.021 0.013	0.032 0.026	0.656 0.650	0.656 0.650	0.05	0.630 0.590	0.630 0.590	0.695 0.685	0.695 0.685	0.048 0.042	0.020	0.057 0.040	0.007	0.007	0.004	0.085	0.085	

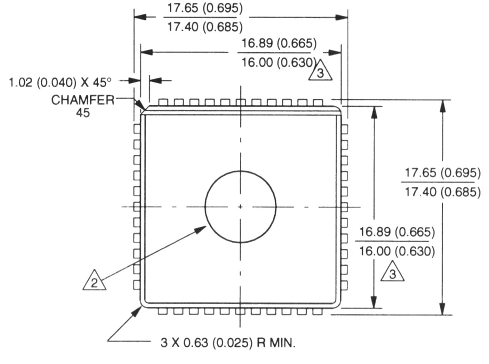
Note

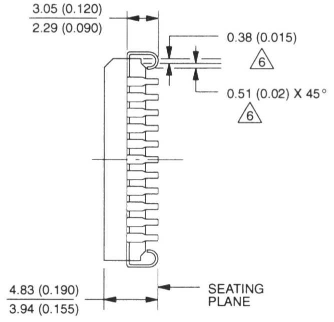
1. Plastic or metal protrusions of 0.01 inches maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT187-2	112E10	MO-047AC				~~92-11-17~~ 95-02-25

Package outlines

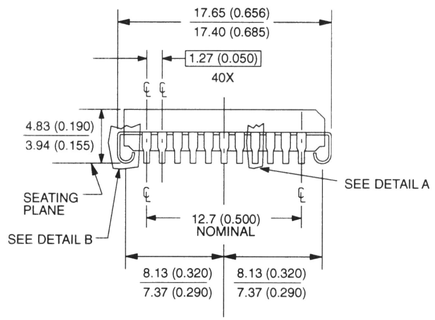
1472A 44-PIN CERQUAD J-BEND (K) PACKAGE

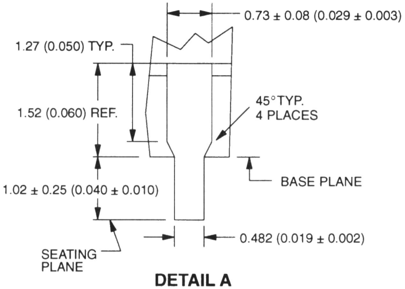

17.65 (0.695)
17.40 (0.685)
16.89 (0.665)
16.00 (0.630) 3
1.02 (0.040) X 45°
CHAMFER
45
17.65 (0.695)
17.40 (0.685)
16.89 (0.665)
16.00 (0.630)
3
2
3 X 0.63 (0.025) R MIN.


3.05 (0.120)
2.29 (0.090)
0.38 (0.015)
6
0.51 (0.02) X 45°
6
4.83 (0.190)
3.94 (0.155)
SEATING PLANE

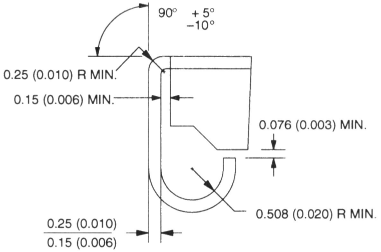
NOTES:

1. All dimensions and tolerances to conform to ANSI Y14.5–1982.
2. UV window is optional.
3. Dimensions do not include glass protrusion. Glass protrusion to be 0.005 inches maximum on each side.
4. Controlling dimension millimeters.
5. All dimensions and tolerances include lead trim offset and lead plating finish.
6. Backside solder relief is optional and dimensions are for reference only.


17.65 (0.656)
17.40 (0.685)
1.27 (0.050)
40X
℄ ℄
4.83 (0.190)
3.94 (0.155)
SEATING PLANE
SEE DETAIL B
SEE DETAIL A
℄ ℄
12.7 (0.500)
NOMINAL
8.13 (0.320)
7.37 (0.290)
8.13 (0.320)
7.37 (0.290)


0.73 ± 0.08 (0.029 ± 0.003)
1.27 (0.050) TYP.
1.52 (0.060) REF.
45°TYP.
4 PLACES
BASE PLANE
1.02 ± 0.25 (0.040 ± 0.010)
SEATING PLANE
0.482 (0.019 ± 0.002)

DETAIL A
TYP. ALL SIDES
mm/(inch)


90° +5° −10°
0.25 (0.010) R MIN.
0.15 (0.006) MIN.
0.076 (0.003) MIN.
0.508 (0.020) R MIN.
0.25 (0.010)
0.15 (0.006)

DETAIL B
mm/(inch)

853-1472A 05854

DATA HANDBOOK SYSTEM

Philips Semiconductors data handbooks contain all pertinent data available at the time of publication and each is revised and reissued regularly.

Loose data sheets are sent to subscribers to keep them up-to-date on additions or alterations made during the lifetime of a data handbook.

Catalogs are available for selected product ranges (some catalogs are also on floppy discs).

Our data handbook titles are listed here.

Integrated Circuits

Book	*Title*
IC01	Semiconductors for Radio and Audio Systems
IC02	Semiconductors for Television and Video Systems
IC03	Semiconductors for Telecom Systems
IC04	CMOS HE4000B Logic Family
IC06	High-speed CMOS Logic Family
IC11	General-purpose/Linear ICs
IC12	I^2C Peripherals
IC13	Programmable Logic Devices (PLD)
IC14	8048-based 8-bit Microcontrollers
IC15	FAST TTL Logic Series
IC16	CMOS Integrated Circuits for Clocks and Watches
IC17	Wireless Communications
IC18	Semiconductors for In-car Electronics
IC19	ICs for Datacommunications
IC20	80C51-based 8-bit Microcontrollers
IC22	Desktop Video
IC23	QUBiC Advanced BiCMOS Bus Interface Logic ABT, MULTIBYTE™
IC24	Low Voltage CMOS & BiCMOS Logic
IC25	16-bit 80C51XA Microcontrollers (eXtended Architecture)

Discrete Semiconductors

Book	*Title*
SC01	Diodes
SC02	Power Diodes
SC03	Thyristors and Triacs
SC04	Small-signal Transistors
SC05	Low-frequency Power Transistors and Hybrid IC Power Modules
SC06	High-voltage and Switching NPN Power Transistors
SC07	Small-signal Field-effect Transistors
SC08a	RF Power Bipolar Transistors
SC08b	RF Power MOS Transistors
SC09	RF Power Modules
SC10	Surface Mounted Semiconductors
SC13	Power MOS Transistors including TOPFETs and IGBTs
SC14	RF Wideband Transistors, Video Transistors and Modules
SC15	Microwave Transistors
SC16	Wideband Hybrid IC Modules
SC17	Semiconductor Sensors

Professional Components

PC01	High-power Klystrons and Accessories
PC06	Circulators and Isolators

MORE INFORMATION FROM PHILIPS SEMICONDUCTORS?

For more information about Philips Semiconductors data handbooks, catalogs and subscriptions, contact your nearest Philips Semiconductors national organization, select from the **address list on the back cover of this handbook**. Product specialists are at your service and inquiries are answered promptly.

OVERVIEW OF PHILIPS COMPONENTS DATA HANDBOOKS

Our sister product division, Philips Components, also has a comprehensive data handbook system to support their products. Their data handbook titles are listed here.

Display Components

Book	*Title*
DC01	Colour TV Picture Tubes and Assemblies Colour Monitor Tubes
DC02	Monochrome Monitor Tubes and Deflection Units
DC03	Television Tuners, Coaxial Aerial Input Assemblies
DC05	Flyback Transformers, Mains Transformers and General-purpose FXC Assemblies

Magnetic Products

MA01	Soft Ferrites
MA03	Piezoelectric Ceramics Specialty Ferrites
MA04	Dry-reed Switches

Passive Components

PA01	Electrolytic Capacitors
PA02	Varistors, Thermistors and Sensors
PA03	Potentiometers
PA04	Variable Capacitors
PA05	Film Capacitors
PA06	Ceramic Capacitors
PA07	Quartz Crystals for Special and Industrial Applications
PA08	Fixed Resistors
PA10	Quartz Crystals for Automotive and Standard Applications
PA11	Quartz Oscillators

Professional Components

PC04	Photo Multipliers
PC05	Plumbicon Camera Tubes and Accessories
PC07	Vidicon and Newvicon Camera Tubes and Deflection Units
PC08	Image Intensifiers
PC12	Electron Multipliers

MORE INFORMATION FROM PHILIPS COMPONENTS?

For more information contact your nearest Philips Components national organizaiton shown in the following list.

Argentina: BUENOS AIRES, Tel. (541)786 7635, Fax . (541)786 9367.
Australia: NORTH RYDE, Tel. (02)805 4455, Fax. (02)805 4466.
Austria: WIEN, Tel. (01)60101 1820, Fax. (01)60101 1210.
Belgium: EINDHOVEN, The Netherlands, Tel. (31)40 783749, Fax. (31)40 788399.
Brazil: SÃO PAULO, Tel. (011)821 2333, Fax (011)829 1849.
Canada: SCARBOROUGH, Tel. (0416)292 5161, Fax. (0416)754 6248.
Chile: SANTIAGO, Tel. (02)77 38 16, Fax. (02)735 3594.
China (Peoples Republic of)**:** SHANGHAI, Tel. (021)326 4140, Fax. (021)320 2160.
Columbia: BOGOTA, Tel. (571)249 7624/(571)217 4609, Fax (571)217 4549.
Denmark: COPENHAGEN, Tel. (032)883 333, Fax. (031)571 949.
Finland: ESPOO, Tel. (9)0-50261, Fax. (9)0-520971.
France: SURESNES, Tel. (01)4099 6161, Fax, (01)4099 6431.
Germany: HAMBURG, Tel. (040)3296-0, Fax. (040)3296 213.
Greece: TAVROS, Tel. (01)489 4339/(01)489 4911, Fax. (01)481 5180.
Hong Kong: KWAI CHUNG, Tel. (852)424 5121, Fax. (852)428 6729.
India: BOMBAY, Tel. (022)4938 541, Fax. (022)4938 722.
Indonesia: JAKARTA, Tel. (021)5201122, Fax. (021)5205189.
Ireland: DUBLIN, Tel. (01)640 203, Fax. (01)640 210.
Israel: TEL AVIV, Tel (9723)6450333, Fax. (9723)493272.
Italy: MILANO, Tel. (02)6752.3302, Fax. (02)6752.3300.
Japan: TOKIO, Tel. (03)3740 5143, Fax. (03)3740 5035.
Korea (Republic of)**:** SEOUL, Tel. (02)709-1412, Fax. (02)709-1415.
Malaysia: KUALA LUMPUR, Tel. (03)757 5511, Fax. (03)757 4880.
Mexico: CHI HUA HUA, Tel. (016)18-67-01/(016)18-67-02, Fax. (016)778 0551.
Netherlands: EINDHOVEN, Tel. (040)783749, Fax. (040)788399.
New Zealand: AUKLAND, Tel. (09)849-4160, Fax. (09)849-7811.
Norway: OSLO, Tel. (22)74 8000, Fax (22)74 8341.
Pakistan: KARACHI, Tel. (021)587 4641-49, Fax. (021)577035/5874546.
Philippines: MANILA, Tel. (02)810-0161, Fax. (02)817-3474.
Portugal: LINDA-A-VELHA, Tel. (01)14163160/4163333,
Fax. (01)14163174/4163366.
Singapore: SINGAPORE, Tel. (65)350 2000, Fax. (65)355 1758.
South Africa: JOHANNESBURG, Tel. (011)470-5911, Fax. (011)470-5494.
Spain: BARCELONA, Tel. (03)301 6312, Fax. (03)301 4243.
Sweden: STOCKHOLM, Tel. (08)632 2000, Fax. (08)632 2745.
Switzerland: ZÜRICH, Tel. (01)488 2211, Fax. (01) 481 7730.
Taiwan: TAIPEI, Tel. (02)388 7666, Fax. (02)382 4382.
Thailand: BANGKOK, Tel. (662)398-0141, Fax. (662)398-3319.
Turkey: ISTANBUL, Tel. (0212)279 2770, Fax. (0212)269 3094.
United Kingdom: LONDON, Tel. (071)590 6633, Fax. (071)636 0394.
United States: RIVIERA BEACH, Tel. (407)881-3200, Fax, (407)881-3300.
Uruguay: MONTEVIDEO, Tel. (02)704 044, Fax (02)920 601.

For all other countries apply to:
Philips Components.
Marketing Communications,
P.O. Box 218,
5600 MD, EINDHOVEN, The Netherlands
Telex 35000 phtcnl, Fax. +31–40–724547.

North American Sales Offices, Representatives and Distributors

PHILIPS SEMICONDUCTORS
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, CA 94088-3409

ALABAMA
Huntsville
Philips Semiconductors
Phone: (205) 464-0111
(205) 464-9101

Elcom, Inc.
Phone: (205) 830-4001

ARIZONA
Scottsdale
Thom Luke Sales, Inc.
Phone: (602) 451-5400

Tempe
Philips Semiconductors
Phone: (602) 820-2225

CALIFORNIA
Calabasas
Philips Semiconductors
Phone: (818) 880-6304

Irvine
Philips Semiconductors
Phone: (714) 453-0770

Loomis
B.A.E. Sales, Inc.
Phone: (916) 652-6777

San Diego
Philips Semiconductors
Phone: (619) 560-0242

San Jose
B.A.E. Sales, Inc.
Phone: (408) 452-8133

Sunnyvale
Philips Semiconductors
Phone: (408) 991-3737

COLORADO
Englewood
Philips Semiconductors
Phone: (303) 792-9011

Thom Luke Sales, Inc.
Phone: (303) 649-9717

CONNECTICUT
Wallingford
JEBCO
Phone: (203) 265-1318

FLORIDA
Clearwater
Conley and Assoc., Inc.
Phone: (813) 572-8895

Oviedo
Conley and Assoc., Inc.
Phone: (407) 365-3283

GEORGIA
Norcross
Elcom, Inc.
Phone: (404) 447-8200

ILLINOIS
Itasca
Philips Semiconductors
Phone: (708) 250-0050

Schaumburg
Micro-Tex, Inc.
Phone: (708) 885-8200

INDIANA
Indianapolis
Mohrfield Marketing, Inc.
Phone: (317) 546-6969

Kokomo
Philips Semiconductors
Phone: (317) 459-5355

MARYLAND
Columbia
Third Wave Solutions, Inc.
Phone: (410) 290-5990

MASSACHUSETTS
Chelmsford
JEBCO
Phone: (508) 256-5800

Westford
Philips Semiconductors
Phone: (508) 692-6211

MICHIGAN
Novi
Philips Semiconductors
Phone: (810) 347-1700

Mohrfield Marketing, Inc.
Phone: (810) 348-5799

MINNESOTA
Bloomington
High Technology Sales
Phone: (612) 844-9933

MISSOURI
Bridgeton
Centech, Inc.
Phone: (314) 291-4230

Raytown
Centech, Inc.
Phone: (816) 358-8100

NEW JERSEY
Toms River
Philips Semiconductors
Phone: (908) 505-1200
(908) 240-1479

NEW YORK
Ithaca
Bob Dean, Inc.
Phone: (607) 257-0007

Rockville Centre
S-J Associates
Phone: (516) 536-4242

Wappingers Falls
Philips Semiconductors
Phone: (914) 297-4074

Bob Dean, Inc.
Phone: (914) 297-6406

NORTH CAROLINA
Charlotte
Elcom, Inc.
Phone: (704) 543-1229

Greensboro
Elcom, Inc.
Phone: (919) 273-8887

Matthews
Elcom, Inc.
Phone: (704) 847-4323

OHIO
Columbus
S-J Associates, Inc.
Phone: (614) 885-6700

Kettering
S-J Associates, Inc.
Phone: (513) 298-7322

Solon
S-J Associates, Inc.
Phone: (216) 349-2700

OREGON
Beaverton
Philips Semiconductors
Phone: (503) 627-0110

Western Technical Sales
Phone: (503) 644-8860

PENNSYLVANIA
Erie
S-J Associates, Inc.
Phone: (216) 888-7004

Hatboro
Delta Technical Sales, Inc.
Phone: (215) 957-0600

Pittsburgh
S-J Associates, Inc.
Phone: (216) 349-2700

SOUTH CAROLINA
Greenville
Elcom, Inc.
Phone: (803) 370-9119

TENNESSEE
Dandridge
Philips Semiconductors
Phone: (615) 397-5053

TEXAS
Austin
Philips Semiconductors
Phone: (512) 339-9945

Synergistic Sales, Inc.
Phone: (512) 346-2122

Houston
Philips Semiconductors
Phone: (713) 999-1316

Synergistic Sales, Inc.
Phone: (713) 937-1990

Richardson
Philips Semiconductors
Phone: (214) 644-1610
(214) 705-9555

Synergistic Sales, Inc.
Phone: (214) 644-3500

UTAH
Salt Lake City
Electrodyne
Phone: (801) 264-8050

WASHINGTON
Bellevue
Western Technical Sales
Phone: (206) 641-3900

Spokane
Western Technical Sales
Phone: (509) 922-7600

WISCONSIN
Waukesha
Micro-Tex, Inc.
Phone: (414) 542-5352

CANADA PHILIPS SEMICONDUCTORS CANADA, LTD.
Calgary, Alberta
Philips Semiconductors/
Components, Inc.
Phone: (403) 293-5969

Tech-Trek, Ltd.
Phone: (403) 241-1719

Kanata, Ontario
Philips Semiconductors
Phone: (613) 599-8720

Tech-Trek, Ltd.
Phone: (613) 599-8787

Montreal, Quebec
Philips Semiconductors/
Components, Inc.
Phone: (514) 424-7320

Mississauga, Ontario
Tech-Trek, Ltd.
Phone: (416) 238-0366

Richmond, B.C.
Tech-Trek, Ltd.
Phone: (604) 276-8735

Scarborough, Ontario
Philips Semiconductors/
Components, Ltd.
(416) 292-5161

Ville St. Laurent, Quebec
Tech-Trek, Ltd.
Phone: (514) 337-7540

MEXICO
Anzures Section
Philips Components
Phone: (800) 234-7381

El Paso, TX
Philips Components
Phone: (915) 775-4020

PUERTO RICO
Caguas
Mectron Group
Phone: (809) 746-3522

DISTRIBUTORS
Contact one of our local distributors:
Allied Electronics
Anthem Electronics
Arrow/Schweber Electronics
Future Electronics (Canada only)
Hamilton Hallmark
Marshall Industries
Newark Electronics
Richardson Electronics
Wyle Electronics
Zeus Electronics

07/06/95

©Philips Electronics North America Corporation 1995 Printed in U.S.A.

Philips Semiconductors - a worldwide company

Argentina: IEROD, Av. Juramento 1992 – 14.b (1428) BUENOS AIRES,
Tel. (541) 786 7633, Fax . (541) 786 9367

Australia: 34 Waterloo Road, NORTH RYDE, NSW 2113,
Tel. (02) 805 4455, Fax. (02) 805 4466

Austria: Triester Str. 64, A-1101 WIEN, P.O. Box 213,
Tel. (01) 60 101-1236, Fax. (01) 60 101-1211

Belgium: Postbus 90050, 5600 PB EINDHOVEN, The Netherlands,
Tel. (31)40 783 749, Fax. (31)40 788 399

Brazil: Rua do Rocio 220 – 5th Floor, Suite 51
CEP: 04552-903 SÃO PAULO-SP, Brazil
P.O. Box 7383 (01064-970),
Tel. (011) 821-2333, Fax (011) 829-1849

Canada: PHILIPS SEMICONDUCTORS/COMPONENTS:
Tel. (800) 234-7381, Fax. (708) 296-8556

Chile: Av. Santa Maria 0760, SANTIAGO,
Tel. (02) 773 816, Fax. (02) 777 6730

China/Hong Kong: 501 Hong Kong Industrial Technology Centre,
72 Tat Chee Avenue, Kowloon Tong, HONG KONG,
Tel. (852) 2319 7888, Fax. (852) 2319 7700

Colombia: IPRELENSO LTDA, Carrera 21 No. 56-17, 77621 BOGOTA,
Tel. (571)249 7624/(571)217 4609, Fax. (571)217 4549

Denmark: Prags Boulevard 80, PB 1919, DK-2300 COPENHAGEN S,
Tel. (032) 88 2636, Fax. (031) 57 1949

Finland: Sinikalliontie 3, FIN-02630 ESPOO,
Tel. (358)0-615 800, Fax. (358)0-61580 920

France: 4 Rue du Port-aux-Vins, BP317, 92156 SURESNES Cedex,
Tel. (01)4099 6161, Fax. (01)4099 6427

Germany: P.O. Box 10 63 23, 20043 HAMBURG,
Tel. (040) 3296-0, Fax. (040) 3296 213

Greece: No. 15, 25th March Street, GR 17778 TAVROS,
Tel. (01) 4894 339/4894 911, Fax. (01) 4814 240

India: Philips INDIA Ltd., Shivsagar Estate, A Block,
Dr. Annie Besant Rd., Worli, BOMBAY 400 018,
Tel. (022)4938 541, Fax. (022)4938 722

Indonesia: Philips House, Jalan H.R. Rasuna Said Kav. 3-4,
P.O. Box 4252, JAKARTA 12950
Tel. (021)5201 122, Fax. (021)5205 189

Ireland: Newstead, Clonskeagh, DUBLIN 14,
Tel. (01)7640 000, Fax. (01)7640 200

Italy: PHILIPS SEMICONDUCTORS S.r.l.,
Piazza IV Novembre 3, 20124 MILANO,
Tel. (0039)2 6752 2531, Fax. (0039)2 6752 2557

Japan: Philips Bldg. 13-37, Kohnan 2-chome, Minato-ku, TOKYO 108,
Tel. (03)3740 5130, Fax. (03)3740 5077

Korea: Philips House, 260-199 Itaewon-dong, Yongsan-ku, SEOUL,
Tel. (02)709-1412, Fax. (02)709-1415

Malaysia: No. 76 Jalan Universiti, 46200 PETALING JAYA, SELANGOR,
Tel. (03)750 5214, Fax. (03)757 4880

Mexico: 5900 Gateway East, Suite 200, EL PASO, TX 79905,
Tel. 9-5 (800)234-7381, Fax. (708)296-8556

Netherlands: Postbus 90050, 5600 PB EINDHOVEN, Bldg. VB
Tel. (040)783749, Fax. (040)788399
(From 10-10-1995: Tel. (040)2783749, Fax. (040)2788399)

New Zealand: 2 Wagener Place, C.P.O. Box 1041, AUCKLAND,
Tel. (09)849-4160, Fax. (09)849-7811

Norway: Box 1, Manglerud 0612, OSLO,
Tel. (022)74 8000, Fax (022)74 8341

Pakistan: Philips Electrical Industries of Pakistan Ltd.,
Exchange Bldg. ST-2/A, Block 9, KDA Scheme 5, Clifton,
KARACHI 75600, Tel. (021)587 4641-49, Fax. (021)577035/5874546

Philippines: PHILIPS SEMICONDUCTORS PHILIPPINES Inc.,
106 Valero St. Salcedo Village, P.O. Box 2108 MCC, MAKATI,
Metro MANILA, Tel. (02)810 0161, Fax. (02)817 3474

Portugal: PHILIPS PORTUGUESA, S.A.,
Rua dr. AntÓnio Loureiro Borges 5, Arquiparque – Miraflores,
Apartado 300, 2795 LINDA-A-VELHA,
Tel. (01)4163160/4163333, Fax. (01)4163174/4163366

Singapore: Lorong 1, Toa Payoh, SINGAPORE 1231,
Tel. (65)350 2000, Fax. (65)251 6500

South Africa: S.A. PHILIPS Pty Ltd.,
195-215 Main Road, Martindale, 2092 JOHANNESBURG,
P.O. Box 7430, Johannesburg 2000,
Tel. (011)470-5911, Fax. (011)470-5494

Spain: Balmes 22, 08007 BARCELONA,
Tel. (03)301 6312, Fax. (03)301 42 43

Sweden: Kottbygatan 7, Akalla. S-164 85 STOCKHOLM,
Tel. (0)8-632 2000, Fax. (0)8-632 2745

Switzerland: Allmendstrasse 140, CH-8027 ZÜRICH,
Tel. (01)488 2211, Fax. (01)481 7730

Taiwan: PHILIPS TAIWAN Ltd., 23-30F, 66, Chung Hsiao West Road,
Sec. 1. Taipeh, Taiwan ROC, P.O. Box 22978, TAIPEI 100,
Tel. (02)388 7666, Fax. (02)382 4382

Thailand: PHILIPS ELECTRONICS (THAILAND) Ltd.,
209/2 Sanpavuth-Bangna Road Prakanong,
BANGKOK 10260, Thailand
Tel. (662)398-0141, Fax. (662)398-3319

Turkey: Talatpasa Cad. No. 5, 80640 GÜLTEPE/ISTANBUL,
Tel. (0212)279 2770, Fax. (0212)282 6707

United Kingdom: Philips Semiconductors Ltd.,
276 Bath Road, Hayes, MIDDLESEX UB3 5BX
Tel. (0181)730-5000, Fax. (0181)754-8421

United States: 811 East Arques Avenue, SUNNYVALE, CA 94088-3409,
Tel. (800)234-7381, Fax. (708)296-8556

Uruguay: Coronel Mora 433, MONTEVIDEO,
Tel. (02)70-4044, Fax (02)92 0601

Internet: http://www.semiconductors.philips.com/ps/

For all other countries apply to: Philips Semiconductors,
International Marketing and Sales, Building BE-p,
P.O. Box 218, 5600 MD, EINDHOVEN, The Netherlands,
Telex 35000 phtcnl, Fax +31-40-724825 (from 10-10-1995: +31-40-2724825)

SCD41 ©Philips Electronics N.V. 1995

All rights are reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner.

The information presented in this document does not form part of any quotation or contract, is believed to be accurate and reliable and may be changed without notice. No liability will be accepted by the publisher for any consequence of its use. Publication thereof does not convey nor imply any license under patent- or industrial or intellectual property rights.

Printed in the USA

5244M/25M/FP/pp520 Date of release: 08-95

Document order number: 9397 750 00219

Philips
Semiconductors

PHILIPS

PHILIPS